Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Hannu Lyytikäinen

Designing Web Services for Location-Aware Mobile Devices

Case: Traffic Monitoring Service

Master's Thesis
Espoo, May 16, 2012

Supervisor: Professor Jukka K. Nurminen, Aalto University
Instructor: Jani Lammi, M.Sc. (Tech.), Gofore Oy

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Hannu Lyytikäinen

Title:
Designing Web Services for Location-Aware Mobile Devices
Case: Traffic Monitoring Service

Date: May 16, 2012 **Pages:** viii + 89

Professorship: Data Communication Software **Code:** T-110

Supervisor: Professor Jukka K. Nurminen

Instructor: Jani Lammi, M.Sc. (Tech.)

Open remote programming interfaces and technologies that enable the development of mashup applications have revolutionized the the way the World Wide Web is used. The emergence of smartphones has provided a new platform for which to build applications that people can use regardless of their location. The location-aware features of smartphones have made it possible for the mobile mashup applications to customize the content they provide for users based on their location.

In this thesis I study how Web services should be designed and implemented so that they would serve location-aware mobile mashup application in the best possible way. I lay down the requirements that this sort of Web service has and then look into different technological and architectural solutions that are available to create a location-aware mobile-friendly Web service interface.

As the practical part of the thesis, I use the knowledge gathered from my theoretical study to implement a new Web service interface for a traffic monitoring system. The new interface is created because the system currently lacks an interface that is mobile-friendly and enables the customization of content based on the location of the user.

To prove that the new interface solves the problem at hand, I implement a mobile application that consumes the new Web service interface. The client application is also used to measure sizes of responses returned by the new API, the time used to process them and what kind of effect location-based optimization has on the API.

Based on the results of the implementation process and the findings of the testing phase, I propose a set of design guidelines that can be applied when developing a Web service interface for mobile location-aware devices.

Keywords: mobile, location-aware, Web service, open data, REST, SOAP, WSDL, Atom, RSS, traffic monitoring

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Hannu Lyytikäinen		
Työn nimi:	Web-palveluiden suunnittelu sijaintitietoisille mobiililaitteille Tapaus: liikenteenvalvontajärjestelmä		
Päiväys:	16. toukokuuta 2012	**Sivumäärä:**	viii + 89
Professuuri:	Tietoliikenneohjelmistot	**Koodi:**	T-110
Valvoja:	Professori Jukka K. Nurminen		
Ohjaaja:	Diplomi-insinööri Jani Lammi		

Avoimet etäohjelmointirajapinnat sekä teknologiat, jotka mahdollistavat mashup-sovellusten kehittämisen ovat mullistaneet tavan jolla käytämme World Wide Webiä. Älypuhelimien yleistyminen on tarjonnut uuden alustan sovelluksille, joita ei ole sidottu mihinkään paikkaan vaan ne kulkevat ihmisten mukana. Älypuhelinten sijaintitietoiset ominaisuudet ovat mahdollistaneet mobiilien mashup-sovellusten sisällön räätälöimisen käyttäjän sijainnin mukaan.

Tässä diplomityössä tutkin, miten Web-palveluita tulisi suunnitella ja toteuttaa, jotta ne parhaalla mahdollisella tavalla palvelisivat sijaintitietoisia mashup-sovelluksia mobiililaitteissa. Esitän vaatimukset, joita tällaisella Web-palvelulla on sekä tutkin millaisia teknologisia sekä arkkitehtuurisia käytäntöjä on olemassa mobiiliystävällisten sijaintitietoisten Web-palvelurajapintojen kehittämiseksi.

Työn käytännön osuudessa käytän teoreettista tutkimustani hyväkseni kehittäessäni uuden Web-palvelurajapinnan liikenteenhallintajärjestelmälle. Uusi rajapinta tarvitaan, sillä järjestelmästä puuttuu etäohjelmointirajapinta, joka mahdollistaisi mobiilit käyttäjäsovellukset, joissa sisältö on räätälöity käyttäjän sijainnin mukaan.

Todentaakseni, että uusi rajapinta ratkaisee olemassaolevan ongelman, toteutan mobiilin käyttäjäsovelluksen, joka käyttää uutta rajapintaa. Käyttäjäsovelluksen avulla myös mitataan rajapinnan palauttamien viestien kokoa, niiden prosessoimiseen käytettävää aikaa sekä sitä millainen vaikutus lokaatiopohjaisella optimoinnilla on rajapinnan toimintaan.

Kehitystyön ja testitulosten pohjalta esitän joukon suosituksia, joita tulisi noudattaa kun kehitetään sijaintitietoisille mobiilisovelluksille tarkoitettua Web-palvelurajapintaa.

Asiasanat:	mobiili, sijaintitietoinen, Web-palvelu, avoin tieto, REST, SOAP, WSDL, Atom, RSS, liikenteenseuranta
Kieli:	Englanti

Acknowledgements

I would like to thank my supervisor, Professor Jukka Nurminen for his valuable advice and devoted attitude that helped and inspired me throughout the process. I would also like to express my gratitude to my instructor Jani Lammi for his technical advice and to Gofore Oy for providing me with the freedom to influence the topic of this thesis.

Helsinki, May 16, 2012

Hannu Lyytikäinen

Abbreviations and Acronyms

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
BEEP	Blocks Extensible Exchange Protocol
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JSONP	JSON with padding
P2P	Peer-to-peer
REST	REpresentational State Transfer
ROA	Resource-Oriented Architecture
RPC	Remote Procedure Call
SOAP	originally defined as Simple Object Access Protocol
UDDI	Universal Description Discovery and Integration
WSDL	Web Service Description Language
XML	eXtensible Markup Language

Contents

Chapter 1

Introduction

The World Wide Web (from now on Web) constantly evolves to new directions. Originally it consisted of static Web sites that people could access with their browsers. This meant that the information on a site could only be accessed by navigating to that site with a browser. Other information had to be gathered from other sites.

From that collection of individual sites the Web has evolved to what we nowadays know as the programmable Web. This means that the resources of the Web are no longer intended only to be consumed by humans but also by computer programs. Nowadays many Web sites combine content from multiple sources. This sort of applications are generally known as *mashups*. The mashup applications and technologies used to create them have evolved from screen scraping to well designed application programming interfaces (APIs) that expose the resources of the Web to applications that can access them, use them and combine them with other resources in ways that would have not been possible with the model of the traditional human consumable Web. The applications that provide a remote interface that can be accessed over the internet are generally known as Web services.

Considering the architectural design of the Web and the applications that run on top of it, a converge of the so called human Web and the programmable Web can be observed. The gap between these two concepts is narrowing as new Web technologies allow developers to create applications that produce information in multiple forms. This means that the information can be accessed in human and machine readable way.

Another factor in the convergence of human Web and programmable Web is the architectural style how Web applications are currently designed. REST[20] is an architectural style that defines a set of constraints for the architecture of a network based application. It has proven itself to be an applicable set of guidelines for a Web based application, regardless of whether

the application is intended to be accessed by humans with browsers or by other applications calling its API.

As open APIs have become popular, mobile devices, especially smartphones, have started to gain foothold as a mashup client platform. The decreased prices of smartphones have made them available to more people and this has increased the popularity of mobile mashup applications. Mobile operating systems have evolved to enable different kinds of new applications. Application stores have made it easier for developers to publish their applications and users to find them. Furthermore, faster mobile internet connections and large displays in mobile devices have enabled the development of new innovative mobile mashup applications.

An interesting new aspect in mobile mashup development is *location-awareness*. It means that mobile client devices can locate themselves and this location information can be used by applications to provide better information for users.

The increasing popularity of location-aware mobile devices has placed new requirements on Web services that are used to gather data for mashup applications. Compared to desktop applications, mobile devices have less bandwidth, limited processing power and smaller displays that set limitations for mobile applications. These limitations induce new requirements for the Web services that are used to create mobile mashup applications. Web services should provide optimized responses in formats that can be transmitted over slow connections and deserialized in the client side in an effective manner. Furthermore, there is no use of the location-aware features of client devices if Web services do not utilize the location information.

This thesis investigates how APIs should be designed so that they would serve mobile location-aware mashup applications in best possible way. The research is done by applying new ideas on a traffic monitoring service.

1.1 Digitraffic

Digitraffic is a system that provides information about traffic fluency on roads. It has been developed by Gofore[1] in cooperation with Infotripla[2] for the Finnish Transport Agency[3] and its main objective is to provide the agency with information about the current and past traffic fluency on different roads in Finland. In addition to traffic data, Digitraffic also provides information about weather conditions on the roads.

[1] http://www.gofore.com
[2] http://www.infotripla.fi
[3] http://portal.liikennevirasto.fi

The current system is used with a Web interface that presents a map of Finland. On this map are drawn roads that are divided into segments, also known as links. The links are drawn with different colors that represent the fluency of traffic on each link. The fluency data is gathered to the system by using infrared cameras that are spread along the monitored roads. The cameras take pictures of cars' license plates and when a car passes two consecutive cameras, the system recognizes that the same license plate has traveled a link and the speed for the car on the link can be calculated. With observations of multiple cars, an average speed can be resolved for each link.

This traffic data is saved in a system called Traffic Data Service (TDS). A background process built in Digitraffic polls the TDS data storage and saves the traffic data into its own database. Figure 1.1 presents the relationship of the cameras, TDC and the actual Digitraffic application.

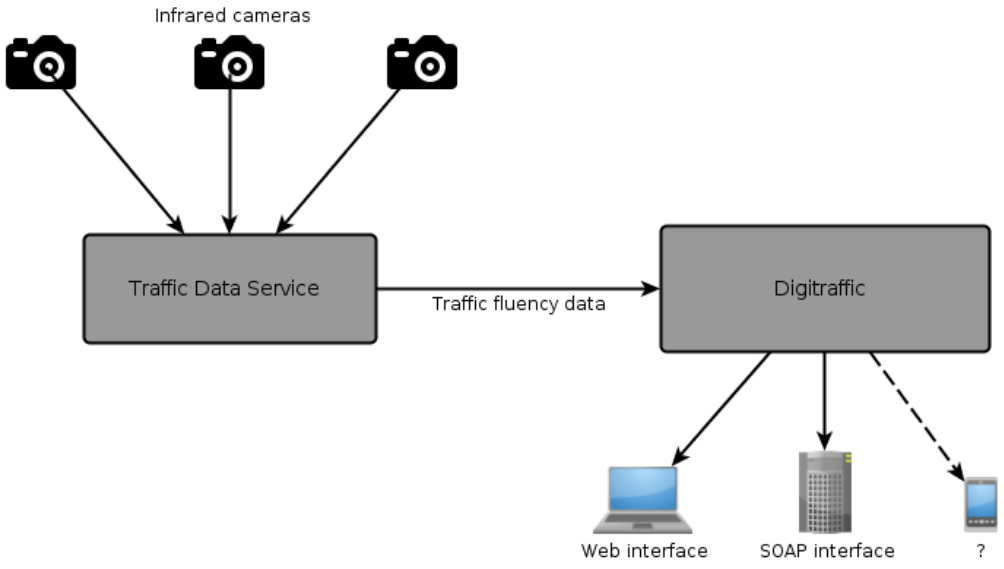


Figure 1.1: Digitraffic

As can be seen in figure 1.1, Digitraffic provides different interfaces for the fluency data. A Web interface is used by professionals from the Finnish Transport Agency to see a map visualization of the current traffic status, road weather information and photos of certain parts of the roads for condition analysis. Traffic fluency data can also be accessed by using a SOAP[31] interface. It provides traffic fluency data for third parties that want to use the information in other applications.

1.2 Problems With the Current API

The SOAP interface was developed back in fall 2007 when Digitraffic was originally created. The reason it was developed was the INSPIRE EU directive[4] which laid down rules about opening spatial data resources to the public. The original purpose of the interface was to provide a low-level API that can be used to fetch the traffic data for enhancement and further use. Any specific purpose for the interface or support for different client devices was not considered. It was just a standardized way to provide traffic information for third parties.

In order to exchange messages, the current Digitraffic remote API uses SOAP which is a Web service technology standardized by the W3C[5]. SOAP relies on XML[12] as message serialization format. The problem with the current low-level nature of the API is that it returns large SOAP messages and SOAP itself is quite a verbose protocol. With the current popularity of mashup applications and the possibilities to build applications for mobile platforms, it is desired that the Digitraffic data can also be provided straight to mobile devices because that way Digitraffic could provide new kinds of possibilities for application developers and end-users. However, mobile devices have different kinds of special requirements like limited processing power and limited bandwidth. Real-time traffic data would be useful for road users. However, people quite rarely bring their desktop computers on road with them but most drivers nowadays carry a smartphone. The problem with the SOAP interface regarding mobile clients is that it provides large response messages because of the verbose SOAP format whereas lighter solutions would be desired.

Furthermore, the current SOAP API does not support any sort of optimization schemes for traffic data queries. When it comes to traffic fluency data, the location of the end-user could be used to optimize queries in order to return smaller messages by just returning data that is valid to the user.

Therefore, a new solution is required for the Digitraffic remote API, to overcome the limitations of the current API. The practical benefit of a mobile friendly Digitraffic API would be that it would enable different kinds of new mashup applications that consume the API. New mobile Digitraffic applications could then enable route planning for users while they are on the road and not only while they are near a desktop computer.

The requirements of the new API design are as follows:

- Enable mobile client application development

[4]http://europa.eu/legislation_summaries/environment/general_provisions/l28195_en.htm
[5]http://www.w3.org

- Be mobile platform independent
- Support lightweight data formats
- Enable query optimization based on user location

1.3 Research Goals

In this thesis I study the architectural guidelines and technologies that ensure the best results when designing and developing remote interfaces that are consumed by location-aware mobile applications. Different possible solutions are evaluated by analyzing their advantages and disadvantages regarding mobile client use. The goal of the thesis is to answer these two questions:

- What are the technologies and design principles that should be used when providing traffic fluency data for mobile devices?
- How location-awareness can be exploited in order to optimize mobile traffic data applications?

Based on the theoretical study, I design and implement a new remote programming interface for Digitraffic. In order to test the new interface, I also implement a mobile client application that consumes the new API. The client application is used to prove the suitability of the technical solution to the problem at hand and to make quantitative measurements about the new solution.

I only concentrate on how the traffic data should be provided to mobile devices. I rule out both the task of collecting the traffic data and the different mathematical models that are used to analyze the data.

The key contribution of this thesis is a design of a new remote programming interface that is mobile-friendly and that exploits the location-aware features of mobile client devices. Other contributions of this thesis are a prototype implementation of the new API design, a mobile client application that consumes the API prototype and the lessons learned from the implementation process.

1.4 Structure of This Thesis

Chapter 2 opens up the concepts related to the topic, presents solutions provided by others to the research problem and explains why the topics of this

thesis need to be studied. Chapter 3 analyzes the different possible technological and architectural solutions. Chapter 4 presents the implementation that is designed based on the findings of chapter 3. Chapter 5 presents the test environment, test results and findings that were gathered from the testing phase. Chapter 6 is for discussion about the results and further development ideas. Chapter 7 provides a conclusion of the thesis, practical benefits and a list the Web service design guidelines that are the main contribution of this thesis.

Chapter 2

Background

Applications that draw content from multiple sources are a growing trend in software development. These mashup applications can fetch data from open APIs and use it in new ways and the location-aware features of mobile client devices can be used to customize mobile mashup application content for users. However, mashup applications that run in mobile devices have certain limitations and special requirements due to the limited resources of mobile handsets. Furthermore, the public sector is nowadays interested in opening public data resources for application developers. This chapter describes how the masuhp paradigm, location-aware mobile client devices and public sector's interest towards open data motivate the research conducted in this thesis. Furthermore, I present what kind of data distribution solutions are implemented in other traffic monitoring services.

2.1 Mashups

Mashup is an application that draws content from multiple sources, processes the information to be used for a new purpose and presents the information in a new user interface. On top of content, mashups can also combine presentation and functionality from other applications[56]. The term originally originates from music where it is used to describe a song that combines two or more pre-recorded songs or compositions[33]. The growing number of open APIs provided by different services have made mashup development more interesting and therefore combining content from multiple sources has become common for Web based applications. Mashup-based design usually emphasizes using Web as platform[33] for applications that serve a special situational and sometimes even short-lived need[56]. Mashup applications rarely produce any data of their own but the value they bring to the existing

data provided by other services is they way they process the data to be used in a new context, how they combine it with content from other sources and the user interface they provide for the processed data[33].

Mashups use different techniques to retrieve their content from the Web. They can use various XML formats, RESTful APIs, feeds like RSS and Atom or even screen scraping[56]. Screen scraping is a method where a markup file intended to be viewed by humans is analyzed an processed by machines. In the context of Web this basically means crawling through HTML documents and extracting information from them to be used in an other context. The downside is that it requires a significant reverse-engineering effort[29].

Another, somewhat less manual, way of developing mashup applications is using designated mashup tools that aggregate and manipulate content from other sources. Applications like Yahoo Pipes[1], IBM Mashup Center[2], Intel Mash Maker[17] and BU Studio[55] can be used to create mashup applications without programming skills. Therefore, these applications enable mashup development for non-technical Web users. However, currently there seems to be problems in the adoption of these kind of systems, for example, in the time of writing this, two big mashup tool projects, Google Mashup Editor and Microsoft Popfly, have been deprecated. This is a shame since a truly liberating tool for end-user could enrich user experience and result in truly innovative systems[42] and therefore enable proper user-driven innovation regarding Web applications.

Mashup application development is often compared to enterprise level integration work where more heavyweight technologies like BPEL, WSCI and Java portlets are commonly used. Using these technologies requires intimate knowledge about schemes and semantics of data sources or business protocol conventions for message exchange. Therefore, building solutions with these technologies has been a rigid and slow process that requires professional programmers and domain experts. Furthermore, the systems tend to be very server-centric and thus do not fully utilize the processing power and storage capabilities of client devices and applications.[56][55] Lighter mashup technologies significantly reduce the complexity and entry barrier of heavier technology stacks[29]. The division between light Web application technologies and heavier enterprise integration technologies seems to be quite clear nowdays. The reason that the heavier technologies are still around is that enterprise processes tend to have a wide set on nonfunctional requirements, such as security and reliability, that are met by enterprise technologies but rarely encountered with mashup applications[56]. Current trend where the

[1]http://pipes.yahoo.com/pipes/

[2]http://www-01.ibm.com/software/info/mashup-center/

mashup technologies vary from traditional integration tool has reduced entry barrier for application development and application complexity in general. It has also enabled mashup applications to be more Web-based, re-usable, lightweight and customer centric.[29]

If we concentrate on mashup applications, we can divide them into to main categories: client-side and server-side mashups. Characteristic for client-side mashups, as the name implies, is that data fetching, processing and application logic is all located in the client application. This can be a native application in a mobile device or a Web site. Server-side mashups on the other hand, retrieve data in the server, process it for the needs of the mashup and then provide it for the client application. In this model the client only acts as a user interface for the data provided by the server. It seems that pure client-side mashups are more common in native applications, for instance in mobile phones. Reasoning for this is provided by the Same Origin Policy, which prevents AJAX calls to foreign domains from Web applications. AJAX (Asynchronous JavaScript and XML) is a set of Web development technologies that can be used to perform asynchronous calls from a Web site to server. It is an essential technique for developing modern responsive Web-based user interfaces. However, the emerging collection of new Web technologies known as HTML5 enables cross-domain AJAX calls, thus making client-side mashups easier to be developed for browsers. This can improve chances for HTML5 to become a popular mashup platform, because it reduces the problems regarding portability to different kinds of devices and operating systems, which appears to be a major problem with native client applications.[6]

Even though mashup development is getting easier and faster, still multiple challenges stand in the way of many good ideas to be realized as innovative applications. Ironically, as content is the ingredient that makes mashup applications so great, it is also the origin of many of the challenges that mashup development faces. For one, there exists many legal issues when consuming content from external sources. This usually means that users of a mashup application have to accept terms of use in order to use the application.[24] Accepting terms of use to use data from unknown sources might raise the barrier for some users to start using an application. In addition to legal issues, problems can also be encountered with data pollution. Many mashups use content that is basically public user input and therefore there is no guarantee on the content the application receives from external sources. Using public input also causes other problems, such as that there is no guarantee of the trustworthiness of data.[29] Therefore, the data may not work as the best content for business critical applications. This sort of lack in accountability has seriously slowed down the adoption of mashups in the enterprise when

compared to the Web, for instance[24]. Another reason preventing wider use of mashups in the enterprise is the lack of a widely dominant model for integrating different components, although Ye et al. propose a new integration pattern for mashups based on component and connector models[55]. Furthermore, lack for common semantics between data elements from multiple sources can cause problems when integrating multiple components in an application[29]. This means that for instance the concept "price" can have different meanings in different systems, like for example in a situation where some definition of "price" includes VAT and some other does not. When developing applications with a long lifespan, the continuity of the support of different APIs can raise problems[33]. Finally, Salo et al. bring out the fact that heavy computing due to information processing in the client end can cause problems with battery life on mobile devices. They also propose a MapReduce-based computing model that can be used to process large amounts of data on the server-side before passing it on to the client[43].

Despite the long list of challenges that developers have to encounter, mashup applications have become a de facto way of combining content from multiple sources in the Web. Websites like ProgrammableWeb[3] collect a comprehensive list of open APIs to be used in mashup applications. The absence of proper integration patterns has not stopped developers from consuming these sources to provide Web users with innovative applications. The emergence of smartphones and tablet PCs has brought new possibilities of running a program in the location context of the user, but it has also placed new challenges for mashup applications. Chapter 2.2 discusses the aspects of mashup development from the point of view of location-aware mobile devices.

2.2 Location-Awareness and Mobility

Lowered prices and constantly improving mobile operating systems have hugely increased popularity of new mobile devices, especially smartphones. Together with improved wireless connections, these devices provide a good platform for mashup development. The fact that they can also use numerous ways to locate themselves brings an interesting new concept into mashup development, location-awareness. It can be used to add the context of location to the user experience of an application. A typical location-aware mobile application is a map application which consumes an external map API to view information retrieved from other APIs on a map. The user location can be used, for example, in a way that the application can show data that is related

[3]http://www.programmableweb.com

to locations near to the whereabouts of the user. In order to determine the location, a mobile device can use IP address, GPS, MAC address in a WLAN network, GSM network or cell phone ID[6].

When designing location-aware mobile applications and the APIs that they consume, a few guidelines have to be remembered. They key is to provide efficient data structures and optimized queries. This kind of design can allow fast queries which then again are essential with location-based services since the the locations and states of different objects can change in a fast pace and therefore data can quickly become obsoleted.[32] Other things to keep in mind are remixability, loose coupling, scalability and ease of deployment[46]. Main issues with mobile devices are usability, connectivity and performance[42] which are caused by small displays, low network bandwidth and limited processing power. Besides connectivity, low bandwidth also affects usability as longer response times from APIs induce longer response times for end-users.

Despite the challenges of mobile application development, mobile devices have become a popular platform for end-user application development. However, when considering Web and the location-aware features of mobile devices, more problems are emerging. The main issue is that even though the devices used by people are location-aware, the Web is not. Basically this means that there occurs mismatches between location concepts in different services. For instance, one service can provide spatial information with radius based semantics while an other service provides same kind of information but with a square based solution.[30] These challenges can lead to tightly coupled integration of map services.

2.3 Open Data

Public governments store huge amounts of data that could be useful for different kinds of mashup applications. The problem standing in the way of innovative applications merging public data is the fact that usually these government administered data storages can only be accessed by authorities.

Open data means information that is produced by public funding and can be openly accessed[37]. This can be any kind of information gathered by a public sector organization. Previously access to this kind of data has been limited but the emergence of varying internet technologies could enable more open data policies[37].

According to Poikolainen et al.[37] more open public data policies could contribute to the following:

- Transparency of democracy and administration

- Creation of new innovations and markets
- Increase efficiency within the government

At the time of writing this, Finland has taken a significant step towards open data policies on government level. A clear indication of this is that providing public data openly has been added to the new government public policy[4]. Here is what it states:

> Data reserves produced with public funding are to be opened to the use of citizens and companies. The goal is to provide digital resources of public sector to be utilized by citizens, companies, communities, authorities and educational as well as research organizations via computer networks in an easily reusable form.

This shows that the number of open data sources can be increasing in the following years also in Finland as it has been in many other countries[37].

2.4 Other Traffic Services

Here I present other traffic surveillance systems and see how they have solved the problems that are studied in this thesis.

2.4.1 Research

A lot of academic research has been conducted in the field of traffic monitoring services. However, most of the research studies different ways to collect the traffic data, making traffic estimations based on the data and different privacy issues that are related to collecting the data but the distribution of the data to third parties has had less attention. Luckily few researches have also covered the topic.

One interesting research project on traffic surveillance is Mobile Millennium[4] which is developed in cooperation by California Center for Innovative Transportation, Nokia Research Center and the University of California in Berkeley. They used GPS enabled smartphones to collect the traffic fluency data straight from the cars that are driving on roads. This system also includes a mobile client application that can be used to monitor traffic fluency. The client applications uses a Navteq[5] Navstreets digital map to visualize

[4]http://traffic.berkeley.edu
[5]http://www.traffic.com

traffic fluency in the client end.[53] Navteq developer resources[6] allow traffic data access for their customers. Currently their traffic data API provides information in XML format and binary feed. Previously they have provided a SOAP API and JavaScript library for traffic data access but these resources are deprecated for new users.

Lin et al.[28] propose a traffic surveillance system relying on a paradigm called Service-Oriented Dynamic Data Driven Application System (SOD-DDAS). Their architecture relies heavily on technologies of the WS stack. Therefore, also they also use SOAP and WSDL to provide access to the traffic data gathered by their system which, like Mobile Millennium, uses moving sensors to gather traffic data. However, their research does not consider consuming the public interface form mobile devices or using location-context to optimize traffic data queries.

An interesting solution to gathering and distributing traffic information is provided by Yang et al.[54] with their peer-to-peer approach. They propose a traffic information system where each vehicle is is presented as a node and information is shared between the nodes without a centralized data center. Each node makes traffic observations and broadcasts them to a selected set of nodes that are elected to be supernodes. The supernodes generate traffic reports from the information they have received. This data is then provided to all the other nodes by using a peer-to-peer network that the vehicles can connect to using WiFi, WiMAX or 3G. In their simulation environment, Yang et al. used Gnutella as their peer-to-peer implementation. This kind of decentralized approach is an interesting aspect in the world of traffic surveillance systems where almost every system seems to rely on some sort of centralized traffic data storage. In this study the simulation environment also included a mobile client application built on a PDA device so this kind of traffic data distribution model has also been tested in mobile context.

In the papers published about traffic systems, the ways to provide the data for client applications vary a lot. Some papers propose interesting new solutions, like Yang et al. with their peer-to-peer proposal. This shows that different means can be used to access traffic information.

2.4.2 Open Traffic Web Services

Here are listed some open APIs that any developer can consume to develop traffic related applications.

511 Driving Times[7] is a RESTful XML-based Web service that provides traffic information in the San Francisco Bay Area.

[6] http://www.nn4d.com/site/global/developer_resources/apis_sdks/p_apis_sdks.jsp

GovHG Data.One[8] is a service provided by the Hong Kong government. It offers traffic speed information, journey time indicators and special traffic news from a RESTful interface in XML format.

Bing Traffic[9] provides traffic incident data as a part of Microsoft's map service. It uses RESTful approach and provides responses in JSON and XML. Bing traffic does not give traffic fluency information but the queries can be optimized with location parameters.

MapQuest Traffic[10] is a RESTful Web service that supports XML, JSON and JSONP for cross-domain requests. Queries can be narrowed down by providing bounding coordinates as parameters and the service returns traffic fluency data as map images that visualize the traffic situation.

In the world of open traffic APIs, the most common way to design interfaces is a RESTful approach with XML or JavaScript Object Notation (JSON)[1] as data format. Therefore, using a RESTful design and lightweight data formats is something that should be looked into when thinking about how the Digitraffic data could be provided to third parties.

2.5 Conclusion

This chapter presented concepts that motivate the research of Web service technologies and the development of new remote API for Digitraffic. The combination of the emergence of mashup applications and the requirements brought by mobile client devices have clearly changed the way remote programming interfaces are used. Location-awareness also brings an interesting new ingredient to the equation. The interest of public sector in open data promotes that publicly owned services, like Digitraffic, should have proper programming interfaces so that the data can be accessed in an appropriate way. Most of the public traffic interfaces in the Web are implemented in some other way than the current Digitraffic Web service interface. Therefore, it is important to study new solutions that could be applied to Digitraffic in order to provide traffic data for mashup applications in a better way.

Chapter 3

Technology Evaluation

This chapter takes a look at different ways to design and implement a new API for Digitraffic. Basic information about the solutions will be given and the advantages and disadvantages of a solution are then compared in contrast to Digitraffic and its requirements. The three selected solutions are REST, WS stack and feeds. Also other possibilities for technical solution are presented but I take a deeper look into the three first ones. The reason for this is that the WS stack is the current implementation technology and REST and feeds seem like the two most prominent challengers to it.

About naming conventions, in this thesis REST means an architectural style, RESTful and ROA are synonyms for an architecture that is designed by following the guidelines of REST. WS stack means the standardized Web service technology stack that consists of SOAP, WSDL, UDDI and the WS-* extension family.

One paradox with comparing WS stack, REST and feeds is that they are completely different concepts. After all, REST is an architectural design, WS stack is a collection of technologies that work together to integrate systems and feeds, in this case Atom and RSS, are individual technologies that can be used to subscribe to content on the Web. However, this bunch of buzzwords is present in the conversation, when developers and companies figure out solutions for open APIs. One possibility would be to compare two widely adopted architectural paradigms, REST and SOA (Service Oriented Architecture). However, the scope of these designs is on completely different abstraction level, so that they could be compared in a rational way. Also SOA targets the question of how enterprise systems are designed, developed and integrated in the long run and therefore it is out of the scope of this thesis, which mainly focuses on mobile mashup applications. Furthermore, the argument "REST vs SOAP" blossoms in the blogsphere and also in the academic papers, so it seems like a adequate question to be discussed here.

Adding feeds to the comparison can be reasoned by the fact that they are essential technologies in today's Web mashup development and there already exists research in applying feeds to spatial services[30].

Different solutions are compared in a way that first the solution and its features are described. Then the applicability of the solution to the research problem is evaluated by studying the research made on this topic. Finally, different solutions are evaluated against the requirements that were proposed for the new Digitraffic API in chapter 1.2.

3.1 REST

In this chapter I will present the REST architectural style for network based computer applications. Chapter 3.1.1 describes the formal definition of the style and chapter 3.1.2 describes how this style is applied in the Web context. Finally I will discuss the applicability of REST to the research problem of the thesis.

3.1.1 REST by Definition

REST is an architectural style for network based computer applications. It defines a set of constraints that describe how an architecture should be designed. This chapter presents REST as it is described in Roy Fielding's dissertation[20].

3.1.1.1 Client-Server

REST style is intended for a system with client-server architectural style. Idea behind this constraint is the separation of concerns. Practically this means that user interface is separated from data storage and business logic. This kind of style improves portability of client applications, simplicity of components and independent deployment of components.

3.1.1.2 Stateless

This constraint means that the server does not hold the state of a session. Therefore, the session data is completely stored and maintained in the client. All the requests sent by the client must contain all the information the server needs in order to understand and process the request. Statelessness improves scalability since servers are able to quickly free resources and, furthermore, it simplifies server side implementation because servers do not need to manage session state between requests.

The disadvantage of statelessness is that it might decrease network performance, since all the information regarding application state has to be included in every request and it can not be stored in the server. Also the server has no control over the implementation of the client which might lead to inconsistent behavior of applications. Then again, in Web it might also be considered as an advantage that different applications can use provided data in different ways.

3.1.1.3 Caching

REST style requires that all server response messages must include a label that implicitly or explicitly tells if the information contained in the message can be cached in the client for later use. Caching can partially or completely remove some interactions, improve efficiency and give better response times to the end user. However, the trade-off of caching is that the information presented by client application might sometimes be invalid or outdated.

3.1.1.4 Uniform Interfaces

REST strongly emphasizes designing uniform interfaces between components. This kind of design is simple and it improves the visibility of interactions. Uniform interfaces help a system to be simpler and component interactions to be more visible. It also enables decoupled components and independent component deployment. The disadvantage of uniformity of interfaces is that it reduces system efficiency. Instead of standardized interfaces, unstandardized interface design that meets the specific needs of an application might provide better efficiency. Then again REST is designed to be a common architecture of the Web, which means it is designed to handle large-grain hypermedia data transfer.

3.1.1.5 Layered System

To improve scalability, REST adds a constraint that addresses layered system architecture. With a layered design, systems can be composed from hierarchical layers which can hide complexity from other components. By abstracting system logic and complexity from other components overall system complexity can be reduced and component independence can be increased. Another advantage of layers is that they can also be used to encapsulate legacy systems and to enable the use of legacy clients for new services.

The trade-off of layers is that they increase overhead and latency in requests over network. In the client end this can appear as longer response times. This latency can be reduced by caching data between the layers.

3.1.1.6 Code-On-Demand

Code-on-demand allows clients to extend their functionality with code downloaded from the server and executed in the client. This kind of design simplifies client implementations and provides a chance to add new features to previously deployed client applications. The disadvantage of code-on-demand is that it reduces visibility. Therefore, it is only an optional constraint in the REST style.

3.1.1.7 Data Elements

Data elements have a key role in REST. They address many of the most important architectural characteristics defined by the design style. In REST components transfer representations of resource states between each other. Representations are serialized in a standardized format that is dynamically selected based on the capabilities and preferences of the recipient and the nature of data.

The most important data element in REST is *resource*. The definition of a resource is anything that can be identified. That might be a Web site, an image, a document, a person, the top ten records in the charts, a set of resources and so on. The important thing to remember is that there has to be a clear way to identify the resource from other resources. Some resources have a static state in a way that their value set stays constant over time. An example of this kind of resource might be "version X" of a document. Other resources have a more dynamic value, for example, "the latest version" of a document. Value of "the latest version" changes over time whenever there is a new latest version but the value of the "version X" stays always the same. However, even if the the value of a resource is dynamic, it's identifier must always stay the same.

A specific resource is identified with a *resource identifier*. REST relies on a naming authority to maintain the validity of the resource identifier over time. Even though an authority maintains the identifier, the author of a resource is expected to choose the resource identifier. This way the resource identifier describes the semantics of the resource.

In REST, resources are passed between components as *representations*. Representations capture the current or intended state of a resource. They consist of data, metadata describing the data and occasionally metadata describing the metadata. Metadata can include representation metadata, resource metadata or control metadata, weather the data describes the representation itself, the resource it captures or the way the representation is to be handled in a component interaction. The data format of a representation

is known as *media type*. It is decided during the interaction according to the capabilities and preferences of the requesting application. Some media types might suite to be rendered and viewed by a user, other might suite better to be processed by computer programs. REST also stresses that media type should enable processing and rendering of the data before it is completely received. This sort of behavior enhances user-perceived performance.

3.1.1.8 Connectors

REST defines multiple connector types to provide activities and transfer resource representations between components. The connector interface design emphasizes generality and clean separation of concerns. This kind of design improves simplicity by providing an abstract interface that hides the underlying implementation of resources and processing mechanisms. General interfaces also allow the implementation to be substituted by another without causing any inconvenience to the clients.

All interactions in REST are stateless. This means that every request holds all the information that is needed to execute that action. This sort of restriction is added to achieve four functions:

1. Removing the need for the server to hold the application state releases resources and improves scalability.

2. Stateless requests allow the requests to be made in parallel so that the server does not need to understand the semantics of the request.

3. Intermediaries may view and understand a request in isolation which may be necessary if services are dynamically rearranged.

4. Every request includes information that might effect the decision of using a cached response in the server.

REST defines five different connector types. The two main types are *client* and *server*. From these two, client is the one that sends the initial request. Server listens for requests, processes them and sends responses.

Third connector type is *cache* connector. It can be used to cache responses to be reused on later interactions. The cache can be located in both client and server interface. In the client end it can be used to reduce network communication and in the server end it can be used to avoid the need of repeating the process of producing a response. A cache can be shared which means that cached responses can also be sent to clients that did not originally send the request that led to the processing and caching of the response.

However, this sort of shared caching can lead to errors if the cached response is not the response that was anticipated by the client.

The two final connector types are *resolver* and *tunnel*. A resolver is used to translate partial or complete resource identifiers into network addresses. In the context of the Web this could be a DNS server for example. A tunnel is needed when network communication has to be relayed over some network boundary. This kind of boundary can be a firewall or a network gateway. Tunnel connector type is added to REST in case some components want to dynamically switch from active component behavior to using a tunnel. Otherwise it could have been left to the network infrastructure to take care of tunnels.

3.1.1.9 Components

Components use the different connectors to complete the REST architecture style. A *user agent* is a component that uses the client connector to access resources. In the Web context the most common user agent is a Web browser which sends requests and renders the responses.

Origin server uses the server connector to control a resource that is requested by user agents. Each origin server provides all the needed retrieval and modification methods to resources that it hosts. The implementation of each method is hidden behind a generic interface. One Web example of an origin serve is an application server.

Last two components are intermediary of type and they both act as a client and as a server. A *proxy* is an intermediary component selected by a client to provide interface encapsulation of other services, data translation, performance enhancement or security protection. A *gateway* provides the same services but is controlled by a server. The main difference between these two is that a proxy is the only one that is selected by the client.

3.1.2 REST Applied to Web

3.1.2.1 Resource-Oriented Architecture

An architecture that follows the principles of the REST architectural style is called *RESTful*. A RESTful architecture is also known as Resource-Oriented Architecture (ROA)[39]. In ROA components, interfaces and interactions are designed and implemented in a RESTful way. This sort of architecture enables scalability, simple system design, generic interface design, independent deployment of components and it places no restrictions on the technologies that are used to implement different components.

3.1.2.2 Resources And URIs

In ROA, resources are identified with Universal Resource Identifiers (URI)[10]. URIs are the "addresses" of the Web so they are the logical identifiers for resources in ROA. In a well designed system, URIs describe the resources they identify, for example:

- `www.example.com/pictures/hello.png`
- `www.example.com/defects/3434`
- `www.example.com/users/8736`

Every URI designates only one resource. On the other hand, one resource can be designated by one or multiple URIs. For example, fetching company sales numbers from `www.example.com/sales/2011/Q1` might result in same byte stream as a resource at `www.example.com/sales/2011_Q1`. Even though these URIs are different they represent the same resource.

Also note that multiple resources can represent the same data. One example of two resources representing the same data might for example be a case, where different versions of a software application are modeled as resources. In that case two valid URIs might be `www.example.com/software/application/versions/1_1_10.tgz` and `www.example.com/software/application/versions/latest`. These two URIs might point to the same file but the ideas behind them are completely different. One points to a specific version of an application and the other points to the latest version of the same application. Therefore, the URIs point to different resources.

3.1.2.3 Addressability

A ROA concept closely related to the URI is addressability. It means that a Web service or a Web site provides every significant piece of information as a resource. So anything that a Web service user might want from a service can be accessed with a URI.

From a Web site point of view addressability can be thought in a way that in an addressable architecture, every site can be bookmarked for later reference. If a system is not built to be addressable, users should download the pages they view so that they could reference them later on.

Addressability also allows completely new ways of consuming resources. For example HTTP proxy services or translation Web sites might take URLs as a parameter when accessing a resource.

3.1.2.4 Statelessness

Just like in the REST definition in chapter 3.1.1, statelessness is also a major part of ROA. In the Web context it means that every single HTTP request is made in isolation from other requests. If addressability means that every meaningful piece of information in a service is provided as a resource and can be accessed with a URI, statelessness means that every server state can be accessed with a URI. This means that also sessions can be provided as resources[27].

In a stateless system, client applications do not need to worry about the order of the requests. The server side does not hold a state, so clients provide all the information needed in every HTTP request. Therefore, there is no need for complex session handling in the server side.

On top of that, statelessness actually makes a Web service more addressable, it also enables more ways to run and maintain a service. The load that is directed to a service can be distributed to multiple servers. In the current trend where horizontal scalability and scaling out are common solutions to growing number of requests, stateless requests fit in perfectly. This way in a system that runs on multiple servers any server can handle any request since previous requests have no effect on new requests.

In order to scale, stateful systems need to replicate every session to every machine the application is running on. Alternatively stateful systems need session affinity, which means that every request in a client session needs to be routed to the same server.

Stateless architecture also makes caching easier. An application can decide weather or not to cache the response of a HTTP request based only on that one response. The application does not have to resolve if the server state has en effect in the response and the response would be different with the next equivalent request.

Some technologies have been proposed in order to add sessions to RESTful architecture, for example, Erenkrantz et al.[18] propose cookies as a way to add sessions to RESTful interactions. However, Richardson and Ruby regard cookies as a non-RESTful technique and that they do no follow the guidelines of ROA[39].

3.1.2.5 Representations

A resource represents a piece of data provided by a Web service. It's data about something but there really is not much else to it. In order for a client to be able to use the data, the service has to provide a representation of the resource. So a representation is sort of the format in which the resource is

presented and it defines how the client processes the data. A resource can be thought of a changing data set but a representation is like on image of a resource in a certain point of time and in a certain format.

One obvious question here is that which representation a server provides for each client request. There are a few ways the client and the server can negotiate the correct representation. A simple way is to include info about the required representation into the URI. Another way is to include the information in the HTTP request headers. Including it in the URI is recommended since this way the URI holds as much information about the requested resource as possible. Also URIs get passed around as links and this way the representation info stays alive longer, whereas HTTP request headers tend to die after the request is committed. The downside in adding the info in the URI is that it increases the number of URIs per one resource which can cause URI dilution. However, the trend in most open APIs nowadays seems to be to add the representation request in the resource URI.

From pure Web service point of view the main question regarding the representation is in which format the requested data is presented in the response. The two most popular formats are JSON and XML. From these two, JSON is currently gaining more popularity in open APIs. One reason for this is that JSON objects are easy to construct and parse in JavaScript and dynamic languages, which are becoming increasingly popular techniques to develop web service clients. Other reason is that JSON is fast to parse, for example when compared to XML[51].

3.1.2.6 General Interfaces

Interface generality is one of the most important principles of REST. In ROA this means that different functions that Web services provide are limited to the HTTP methods: GET, PUT, DELETE, POST, HEAD and OPTIONS. These methods are used to access resources in RESTful systems. Following examples use a blog service `www.fooblog.com` to illustrate the function of each HTTP method in ROA.

GET is used to retrieve data. Sending a GET to a server returns a representation of some specific resource. For example, a client can retrieve a blog post with id 1234 from the blog service by sending a GET request to `www.fooblog.com/blogs/myblog/posts/1234`.

PUT creates a resource. If our blog service identifies blog posts by their topic, a client can create a new blog post by sending a PUT request that contains a representation of a blog post to `www.fooblog.com/blogs/myblog`

/posts/post-about-my-day. PUT can also be used to modify existing resources. By sending a PUT request to a URI of an existing resource, it can be overridden with new data. For example, to update blog post, a client can send a PUT containing the updated post to the existing resource www.fooblog.com/blogs/myblog/posts/post-about-my-day.

POST, just like PUT, also has two functions in ROA and one of them is adding new resources. The difference between adding new resources with POST and PUT, is that with PUT the client has control over the identifier of the new resource. Like we saw in the PUT example, the client gets to decide the identifier of the new post (post-about-my-day). With POST, the server is in charge of assigning an identifier to the new resource. For example, a POST to www.fooblog.com/blogs/myblog/posts creates a new resource where the id can for example be the database key. If 1234 was the id that the server resolved for the new blog post, the URI of the created resource would be www.fooblog.com/blogs/myblog/posts/1234. The other function of POST is to append an existing resource with additional information. In the blog service example this could for example mean adding comments to existing blog posts. If a client uses POST to send some data to www.fooblog.com/blogs/myblog/posts/1234, that data would be appended to the post as a comment. To clear the relationship between PUT and POST, their use cases have been listed in table 3.1.

DELETE removes resources. To remove blog post with id 1234, DELETE request is sent to www.fooblog.com/blogs/myblog/posts/1234.

HEAD returns metadata about a resource without downloading a complete representation of the resource. In other words, HEAD provides an easy way of checking if there exists a resource that corresponds to some URI. Especially in mobile devices clients can save bandwidth and reduce power consumption by using HEAD instead of GET. With the blog service this means that clients can check if a specific post still exists by sending a HEAD to www.fooblog.com/blogs/myblog/posts/1234.

OPTIONS lets clients know what methods does a server provide for different URIs. For example sending an OPTIONS request to www.fooblog.com /blogs/myblog/posts/1234 would return GET, PUT, POST, DELETE and HEAD but sending the request to www.fooblog.com/blogs/myblog would only return GET and HEAD. Also different header fields in the HTTP request can affect the set of possible operations returned by an OPTIONS call. For instance, sending a proper value in Authorization header field can in-

crease the number of possible actions from read-only operations like HEAD and GET to resource altering operations like PUT, POST and DELETE.

Table 3.1: PUT and POST actions

Resource	**PUT (new resource)**	**PUT (existing resource)**	**POST**
/blogs	N/A (resource already exists)	No effect	Create a new blog
/blogs/myblog	Create a new blog	Modify this blog's settings	Create a new blog post
/blogs/myblog /posts/1234	N/A	Edit this blog post	Add a comment to this blog post

When committing an action in a resource-oriented system, the operation should always be defined by the HTTP action and the scope of the operation should be defined by the URI. Many system designs fail here by using GET action for multiple different operations. However, correct use of HTTP methods is the key to a uniform interface design which is important when building systems for the Web scale.

3.1.2.7 Method Safety

In addition to correct method behavior, method safety also has an important role in ROA. Methods that have no effect on the state of any resource in the server side are called *safe*. GET and HEAD are generally safe HTTP methods. Clients can safely commit these methods to APIs as many times as they like and they can rely on that the resources behind the APIs hold their state. However, it is to be noted that even though there would be no changes in resource state, these methods can have some side effects. Some URIs can work as hit counters that count the number of invocations in Web services and most services log invocations that are made to their APIs. So safe operations can have some effects in services but the main idea is that they do not change the actual resources.

An other term closely related to method safety is *idempotence*. Idempotent operations can be invoked one or more times and the effect is always the same as if the operation had been invoked only once. The term originally describes mathematical operations so it can easily be demonstrated with a math example. In math, multiplying with zero is an idempotent operation: $5 \times 0 \times 0 \times 0$ has the same result as 5×0. So in ROA, invoking an idempotent action on a resource for the second time leaves the resource in the same state as invoking the action for the first time. From the HTTP method catalog,

PUT and DELETE are idempotent. For instance, a client uses DELETE to delete a resource, the resource is removed. When the client uses DELETE to remove the resource again, the resource is still just as removed as it was after the first invocation. If a client uses HTTP PUT to create a new resource, the resource is still the same when a new PUT with the same data is invoked to that URI. Respectively, updating a resource with PUT leaves the resource in the same state regardless of the number of method invocations.

Safety and idempotence of interface functions have a major role in RESTful API design and they are one of the main advantages of the uniform interface paradigm. Correct use of HTTP methods in interfaces, allows clients to know which operations have no effects to the state of a resource and which operations should be invoked only when the state of a resource should be altered.

3.1.3 Applying to Digitraffic

3.1.3.1 Advantages

To figure out how RESTful design principles fit into the case of Digitraffic, we have to lay out the pros and cons of RESTful architecture in a system that is aimed for location-aware mobile handsets.

The qualities that we are looking for are good scalability, optimized performance and also message format and platform independence . The entry barrier for client application development should be low as well as the development work should be relatively easy. Of course the interface should also enable the development of complex client applications but the initial learning curve isn't desired to be too steep.

The RESTful approach of ROA meets these requirements particularly well. Client development for this kind of system requires minimal tooling and also the client testing is easy since the most simple functionalities can be tested with a Web browser[36]. Since no heavy enterprise tools are needed, to start developing client applications the development can be done basically with any kind of setup which lowers the barrier for developing client applications.

The general interface design of RESTful systems increases loose coupling between system components, unlike RPC-based system design which usually result in tight coupling[47]. The use of HTTP methods for the possible actions restricts the number of possible actions to be performed[36] but general operations take the whole concept of interface contract out of the client-server interaction[48]. This means that client-side developers do not have to study the methods that different services provide as the methods are always the

same. Everything the client developers need to worry about is the data contract which defines what is included in the request and response messages. General interfaces where actions are limited to HTTP methods also help clients to figure out what the operations do to the resources and if invoking the actions can have irreversible implications[26]. Apart from POST, every HTTP method used in RESTful interface design is either safe or idempotent. Therefore, if an operation fails to return a desired response, it can always be invoked again[39] without a fear of changing resource state in the server side. Also the generic interface design tends to reveal less about the underlying technologies and implementation than specific interfaces. This means that changes to the implementation can require changes also to the interface which can lead to expensive modifications to client applications[48].

The fact that the Digitraffic API is also intended to be used by mobile clients, rises new requirements. Mobile devices have strict bandwidth, memory, processing power and battery life constraints compared to desktop clients [51]. This means that an easily processable lightweight payload format would be optimal for resource representations. RESTful architecture allows any kind of media format to be used as a representation of a resource. Whereas for example SOAP is limited to using XML in message serialization, RESTful service can use lighter representation formats, like JSON. Compared to XML, JSON provides smaller message size, optimized performance[36][51] and therefore enhanced response times for the end user. Erenkrantz et al. found that querying a JSON API can be up to six times faster that querying a SOAP API[18]. Decision on an API interaction format can also be affected by the technologies used to implement the client applications. JSON is based on a subset of the JavaScript[1] programming language, which makes parsing and constructing JSON objects very easy in JavaScript. JavaScript is a popular technology for implementing browser-based applications and browser is currently the platform for many mashup applications in mobile and desktop devices. Furthermore, JSON is well supported by dynamic programming languages that are gaining popularity in server side development of Web applications. RESTful APIs have means to provide support for response caching[36] which reduces the amount of needed requests to be sent by clients to the server. As Fielding points out, the least bandwidth-consuming request is the one that is not made[20] so this sort of behavior reduces the use of bandwidth which can be observed as better response times for the end user as well as improved battery life.

On top of all, RESTful interface design provides features that help building scalable systems. Stateless resources allow load balancing and clustering[36]. RESTful design principles are appropriate for horizontal scaling where resources can be added just by adding more server machines to process re-

quests. Scaling stateful systems would mean duplicating the session to each server or making sure that all the requests of a session are processed by the same machine. A stateless system provides all the information needed about the state of a resource in the resource identifying URI so any server in the back-end system can process any request.

As a conclusion, the advantages of RESTful interface design almost all count as advantages of an interface that is designed to be consumed by mobile devices. Message format and platform independence enable client application development for multiple different mobile platforms and also for mobile browsers. Stateless design works well with mobile clients because the connection to the server can be lost from time to time especially in the case of Digitraffic where the mobile client devices are usually moving in vehicles and therefore vulnerable to connection breaks. That is why it is an advantage that there is no session to be lost. For the same reason method idempotence and safety are valuable features in mobile environment because they allow resending requests again in the case of possible connection problems. Furthermore, a location-aware system can be implemented in a RESTful way as the location is always checked in the client end and never saved in the server-side.

3.1.3.2 Disadvantages

Despite all the good qualities of RESTful approach to Web services, there are also some use cases where the RESTful design does not cover all requirements. Common criticism on RESTful Web services is the lack of agreed standards regarding service interface description and service discovery. Furthermore, RESTful interface design is problematic for systems that are not content-centric[18]. There is no trivial way to model a process-centric system with resources. Also the constraint of statelessness can be regarded as a downside because in business-critical applications it necessarily leaves some of the business-critical processing to the client applications[27].

RESTful Web services lack a way of formally describing their interfaces. APIs are commonly described with informal HTML documents but there are no widely adopted standards about the formal description of RESTful Web services. This also means that there is no formal description of the data types used with requests and responses. Even though there is no widespread technology to describe RESTful interfaces, multiple technologies have been proposed to be used for this task. Some of the proposed description technologies like WRDL[38], NSDL[50], SMEX-D[11], Resdel[15], RSWS[44] and WDL[35] seem to be more or less ad-hoc intentions to solve some specific problem, but have not been updated or developed in years. The most promi-

nent proposals currently are hRESTS[25] and WADL[22].[26] Pautasso et al. also observe that machine processable interface descriptions would enable compiler level check for client applications[36]. However, the most popular way to describe a RESTful interface is an HTML document that defines what methods can be used to which resource and what kind of data formats and structures are used in requests and responses.

There has also been some criticism on the general interface paradigm of REST. Firstly, the lack of formal interface descriptions leaves clients in uncertainty about which methods can be applied to which resources. Secondly the use of all the HTTP methods can be problematic since some firewalls are only open to GET and POST by default. Also most of the methods carry all the parameters in the URI which sets size limitations for the parameters because currently most implementations limit the size of the URI to 4 kilobytes.[36]

Finally, the main concern of RESTful systems and the ROA architecture is its ability to describe systems that provide business-critical services. Mainly this is because of the lack of best practices and agreed standards for discovering new services, orchestrating the services and describing the processes they handle. The REST definition does not include service security and reliability, not to talk about end-to-end security and other quality of service features.[26][36][26] The absence of these qualities have prevented the wider adoption of RESTful system design in enterprise system development. However, none of these features are required when developing mashup applications. They don't promote anarchic scalability, low entry barrier or loosely coupled components, all of which would be desired qualities when building a mobile friendly interface for a system like Digitraffic.

The disadvantages of RESTful system design mostly concern the lack of support for enterprise system integration. The fact that there is no formal way to model business processes is not necessarily a disadvantage when developing a scalable system for mobile devices.

3.2 WS Stack

This chapter presents a Web service technology stack that consists of three main standards: SOAP, WSDL and UDDI. These technologies, also known as the first generation Web service standards[19], are maintained by organizations like W3C and OASIS[1] and they cover the main aspects of implementing, providing and consuming Web services. The purpose of each technology is

[1] http://www.oasis-open.org

as follows:

SOAP Messaging protocol that defines how different components communicate with each other.

WSDL Description of services so that clients know how services are supposed to be invoked.

UDDI Discovery of new services.

Figure 3.1 presents the relationships between the three technologies. From it can be seen how the server, ie. the actual Web service, publishes the service endpoint in an UDDI registry and how the client finds the service from the registry. The service describes itself to the client with WSDL and the client and the server then communicate using SOAP.

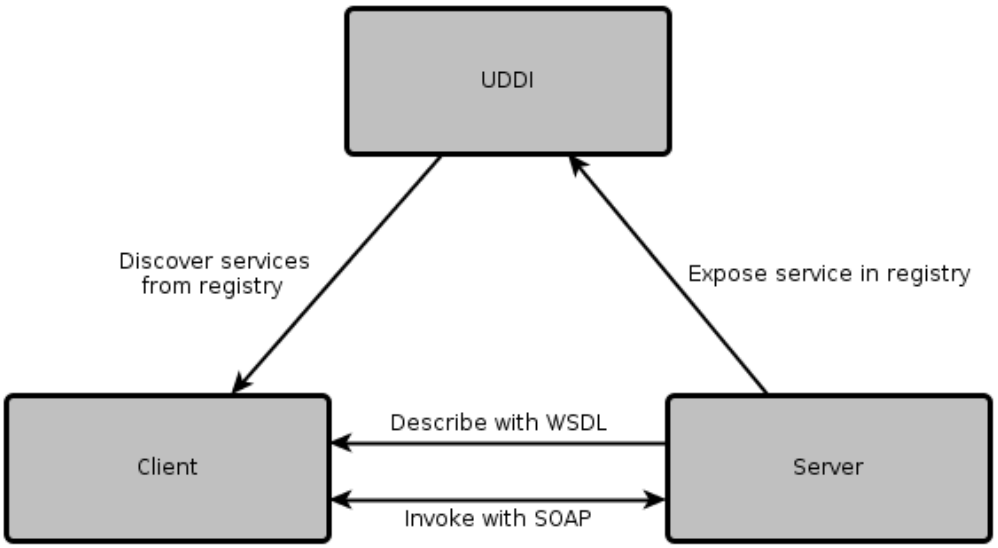


Figure 3.1: Web service technology stack

SOAP, WSDL and UDDI all have released multiple versions that cover varying selection of features and have seen varying levels of adoption. To ensure interoperability of these technologies, the OASIS Web Services Interoperability Organization (WS-I) Member Section[2] provides a set of best practice guidelines called WS-I Basic Profile, which defines the different versions of Web service technologies that should be applied together. In this

[2]www.ws-i.org

thesis I will cover the versions of SOAP, WSDL and UDDI that are mentioned in the latest WS-I Basic Profile version, 2.0[14].

3.2.1 SOAP

SOAP[31] (previously stood for Simple Object Access Protocol) is an XML based one-way message exchange paradigm standardized by the W3C. It is used to describe messages that perform procedure calls between remote systems. Originally it was created by Microsoft and later on Developmentor, IBM, Lotus and UserLand have also committed to the development work[16]. SOAP is independent of platform, programming language and transport protocol. Regardless of SOAP's one-way nature, also more complex messaging patterns can be implemented by combining one-way exchanges and the features of an underlying protocol and/or application-specific information.

3.2.1.1 Messages

A SOAP message is an XML document that has a root element called *envelope*. The envelope element's sub-elements are optional *header* element and mandatory *body* element. The structure is modeled in figure 3.2.

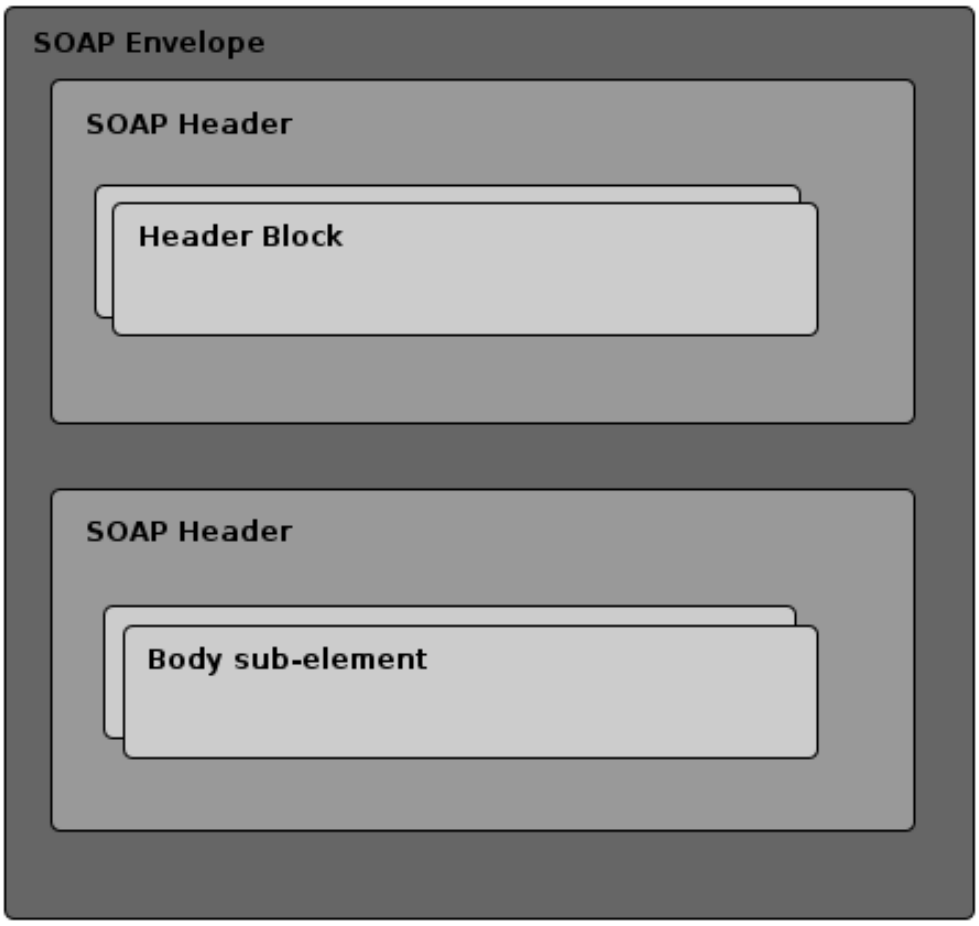


Figure 3.2: Soap message structure

The header element contains meta information about the message. It plays a major role in many architectures and even though its use is optional, it is rarely left out due to to the extension possibilities it provides. The information in the header element is divided into parts that are called *header blocks*.[19]

Header blocks provide SOAP with message independence and they increase extensibility of the protocol. They include rules and instructions on how a SOAP message should be routed and processed when it comes into contact with any system component. Examples of information contained in a header blocks are processing instructions for intermediaries, routing or work flow information, security measures, reliability rules, correlation infor-

mation and also context and transaction management information. This way header blocks can provide a large amount of accessory information on how to transmit and process message content. In addition to processing and routing information, header blocks also provide extensibility for SOAP messages. In fact, practically all WS-* extensions (discussed in chapter 3.2.4) are implemented in header blocks.[19]

The body element is the actual payload of the message and it holds the main end-to-end information of the message.

```
<env:Envelope
    xmlns:env="http://www.w3.org/2003/05/soap-envelope">
    <env:Header>
        <m:specialRequest xmlns:m="http://www.example.com/
           traffic/specialRequest"
          env:role="http://www.w3.org/2003/05/soap-envelope/
             role/next"
           env:mustUnderstand="true">
           <m:priority>HIGH</m:priority>
        </m:specialRequest>
    </env:Header>
    <env:Body>
        <p:TrafficFluencyRequest
    xmlns:p="http://www.example.com/traffic/fluency">
      <p:coordinates>
    <p:latitude>60.190780088420645</p:latitude>
    <p:longitude>24.945144653320312</p:longitude>
      </p:coordinates>
  </p:TrafficFluencyRequest>
    </env:Body>
</env:Envelope>
```

Example 3.1: SOAP message

Example 3.1 shows a SOAP message that is sent to a traffic data service, just like Digitraffic. It has the `env:Header` and `env:Body` elements inside the top level element `env:Envelope`. The `env:Header` element includes information about how to process the message. The `env:role` attribute in the `m:specialRequest` element and its value indicate that the `m:specialRequest` element must be processed in the next intermediary on the path instead of processing it in the final receiving node. The `env:Body` element contains the actual payload of the message. It is processed by the final receiving SOAP node. In this example the `env:Body` element contains coordinate information that is sent to a traffic service in order to receive traffic fluency data around the location defined by those coordinates.

3.2.1.2 Nodes

System components that are responsible for sending, receiving, forwarding and processing SOAP messages are called *SOAP nodes*. Nodes are labeled with node types based on the role of the node in a specific message exchange.

SOAP sender SOAP node that transmits a message

SOAP receiver SOAP node that receives a message

SOAP intermediary SOAP node that receives and transmits a messages, and optionally also processes the message prior to passing it on

Initial SOAP sender First SOAP node to send a message in a given message exchange

Ultimate SOAP receiver Last SOAP node to receive a message in a given message exchange

The SOAP intermediaries can also be divided into *forwarding intermediaries* and *active intermediaries*. Forwarding intermediaries can process a message according to processing logic described in header blocks of the message before transmitting the message to the following SOAP node. Active intermediaries do not limit the processing they perform to the processing logic described in the headers but also might perform other processing as well.[19]

3.2.1.3 Message Exchange

SOAP is a messaging framework which can be used to exchange XML-based messages between a SOAP sender and a SOAP receiver. Complex use-cases can be achieved by combining these kind of request-response exchanges to more complex transactions. Messages can also be modeled as *remote procedure calls* (RPC) in case some sort of programmatic function needs to be invoked by the exchange. Even though the RPC-nature of SOAP is often emphasized, it is also important to remember that SOAP can also be used for a simple request-response message exchange.

Message exchanges that use SOAP as a messaging framework to send and receive XML-based messages are called *conversational message exchanges*. Following example presents a response to the message in the example 3.1.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-
   envelope" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <env:Header />
   <env:Body>
      <p:TrafficFluencyResponse xmlns="http://www.example.com
         /traffic/fluency" xmlns:p="http://www.example.com/
         traffic/fluency">
         <p:timestamp>
            <p:utc>2008-10-09T10:06:49Z</p:utc>
         </p:timestamp>
         <p:laststaticdataupdate>2008-09-30T21:00:00Z</p:
            laststaticdataupdate>
         <p:linkdynamicdata>
            <p:linkstat>
               <p:linkno>0</p:linkno>
               <p:measurementtime>
                  <p:utc>2008-10-09T10:06:00Z</p:utc>
               </p:measurementtime>
               <p:midspeednow>54.108</p:midspeednow>
               <p:fluencyclassnow>5</p:fluencyclassnow>
            </p:linkstat>
            <p:linkstat>
               <p:linkno>1</p:linkno>
      ...
            </p:linkstat>
          </p:linkdynamicdata>
      </p:TrafficFluencyResponse>
   </env:Body>
</env:Envelope>
```

Example 3.2: SOAP response

In example 3.2 again the `env:Body` element contains the primary information processed by the application. Here it includes traffic data returned in response to message described in example 3.1.

More complex functionalities where the request is intended to invoke some action at the SOAP receiver can be implemented by modeling the messages as RPCs. With RPCs the request message includes the method that is invoked and the information that is given as parameter to the method. According to [31], the following information is needed in order to perform an RPC invocation:

1. The address of the target SOAP node
2. The procedure or method name

3. The parameters that are required by the method being invoked
4. A clear separation between the arguments that define the target of the RPC and the information about how the message should be processed.
5. The message exchange pattern and the HTTP method that are to be used.
6. Optionally, data which may be carried as a part of SOAP header blocks.

The receiving SOAP node can provide this information by using a formal service description language, like WSDL, which is described in chapter 3.2.2.

3.2.1.4 Protocol Binding

SOAP messages can be carried between nodes by using an underlying protocol. The specification that defines how a protocol is used to transmit the SOAP message is called a *SOAP binding*. The binding is not required to be the same through out the whole passage of a message. If the message goes through intermediaries on its way, each intermediary can use a different carrier protocol to pass on the message.

In addition to providing a carrier platform for the SOAP messages, the underlying protocol also provides mechanisms to enable additional features for SOAP messages. These features, also known as extensions, can add extra functionality to the basic SOAP specification. If a feature is not provided by an underlying protocol, it can also be built into to SOAP header blocks. Features carried this way in the SOAP message are also called *SOAP modules*. Since every intermediary on the way from the sender to the receiver can use different carrier protocol, all the hops between nodes might not support the same set of features. The missing features in underlying protocols can be compensated by including the missing features in SOAP modules. Additional SOAP features are discussed in more detail in chapter 3.2.4. Furthermore, it's for the SOAP binding to define which features of an underlying protocol are enabled for the SOAP applications. Also the message exchange pattern must be specified in the biding.

The most commonly used protocol binding used with SOAP is the HTTP binding. The SOAP specification also contains a description of an email binding but it's presented more as a mere example of an alternative carrier protocol to HTTP.

3.2.2 WSDL

Web Services Description Language (WSDL)[13] is an XML-based language used to describe Web services and their functionalities. It can be used to model the operations and data types used by a service and how the service communicates with other components. WSDL provides an abstract description of a service interface. It is independent of message formats and network protocols, however, it is bound to a certain protocol in order to provide an endpoint for a service. A common way of accessing WSDL is binding it to SOAP that is transmitted over HTTP.

A WSDL document consists of the following elements:

Types Data type definitions described by some type system (XSD[49] for example)

Message Describes the data that is being communicated

Operation Describes an action that is provided by the service

Port Type Lists the operations that are supported by on or more endpoints

Binding Defines the protocol and data format that are used

Port Service location described as a network location and a protocol binding

Service A collection of service endpoints

A WSDL document can be observed in example 3.3. It describes an interface that can be used to access traffic fluency information. The `types` element provides data types `TrafficFluencyRequest`, `TrafficFluencyResponse` and `LinkStatType` that are used to provide traffic fluency information to clients that consume the service. The two `message` elements describe the messages that are used to communicate with the service and the messages are then used by the `GetTrafficFluency` operation. The operation is listed under the `TrafficFluencyPort portType` element. The `portType` is then bind to SOAP protocol and so on.

```
<definitions name="TrafficFluencyProvider"
targetNamespace="http://example.com/trafficfluency.wsdl"
          xmlns:tns="http://example.com/trafficfluency.wsdl"
          xmlns:xsd1="http://example.com/traffic/fluency.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
```

```
      <schema targetNamespace="http://example.com/traffic/
         fluency.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema"
               xmlns:p="http://www.example.com/traffic/
               fluency">
    <element name="TrafficFluencyRequest">
          <complexType>
              <all>
                  <element name="latitude" type="string"/>
                  <element name="longitude" type="string"/>
              </all>
          </complexType>
          </element>
  <element name="TrafficFluencyResponse">
          <complexType>
              <sequence>
                  <element name="laststaticdataupdate" type
                     ="xsd:dateTime" />
                  <element name="linkdynamicdata">
                      <complexType>
                          <sequence>
                              <element maxOccurs="unbounded
                                 " name="linkstat" type="p:
                                 LinkStatType"/>
                          </sequence>
                      </complexType>
                  </element>
             </sequence>
         </complexType>
       </element>
       <complexType name="LinkStatType">
          <sequence>
             <element name="linkno" type="xsd:
                nonNegativeInteger"/>
             <element name="measurementtime" type="xsd:
                dateTime"/>
             <element name="journeytimenow" type="xsd:
                string"/>
             <element name="fluencyclassnow" type="xsd:
                nonNegativeInteger"/>
          </sequence>
       </complexType>
     </schema>
  </types>
 <message name="TrafficFluencyInput">
    <part element="xsd1:TrafficFluencyRequest" name="body"
       />
 </message>
 <message name="TrafficFluencyOutput">
```

```
      <part element="xsd1:TrafficFluencyResponse" name="body"
         />
   </message>
    <portType name="TrafficFluencyPort">
        <operation name="GetTrafficFluency">
           <input message="tns:TrafficFluencyInput"/>
           <output message="tns:TrafficFluencyOutput"/>
        </operation>
    </portType>
    <binding name="TrafficFluencyBinding" type="tns:
       TrafficFluencyPort">
        <soap:binding style="document" transport="http://
           schemas.xmlsoap.org/soap/http"/>
        <operation name="GetTrafficFluency">
           <soap:operation soapAction="http://example.com/
              GetTrafficFluency"/>
           <input>
               <soap:body use="literal"/>
           </input>
           <output>
               <soap:body use="literal"/>
           </output>
        </operation>
    </binding>
    <service name="TrafficService">
        <documentation>Traffic Service</documentation>
        <port name="TrafficFluencyPort" binding="tns:
           TrafficFluencyBinding">
           <soap:address location="http://example.com/
              trafficfluency"/>
        </port>
    </service>
</definitions>
```

Example 3.3: WSDL service description

3.2.3 UDDI

The Universal Description, Discovery and Integration (UDDI)[9] completes the Web service stack by providing an XML-based protocol for publishing and discovering Web services. An UDDI registry can be public, revealing services to all interested clients in the internet or it can be private, exposing services internally inside an organization. Registries' main functionality is to provide data and metadata about Web services. Therefore, they provide means to classify, catalog and manage services[2]. UDDI is standardized by the Organization for the Advancement of Structured Information Standards

(OASIS)[3].

OASIS presents the following as the main use cases for UDDI[2]:

- Publish information about Web services
- Find publicly or privately published Web services against a certain criteria
- Provide a specification of security and transport protocols that are supported by a Web service and also provide information about parameters that are required by a service
- Insulate applications from failures in invoked services

The UDDI data model consists of four different data types: *businessEntity*, *businessService*, *bindingTemplate* and *tModel*. They provide information about Web services so that service consumers know which services answer to their needs. The main purpose of each data type is as follows:

businessEntity Information about a business or an organization that provides a service

businessService Information about a Web service

bindingTemplate Technical details of a service

tModel Information about how the service behaves, what conventions it follows and what standards it complies with

The relationships between different data models are presented in figure 3.3. It shows that each businessEntity can contain multiple businessServices and that each businessService can contain multiple bindingTemplates. Furthermore, each bindingTemplate can contain references to multiple tModel entites.

[3]www.oasis-open.org

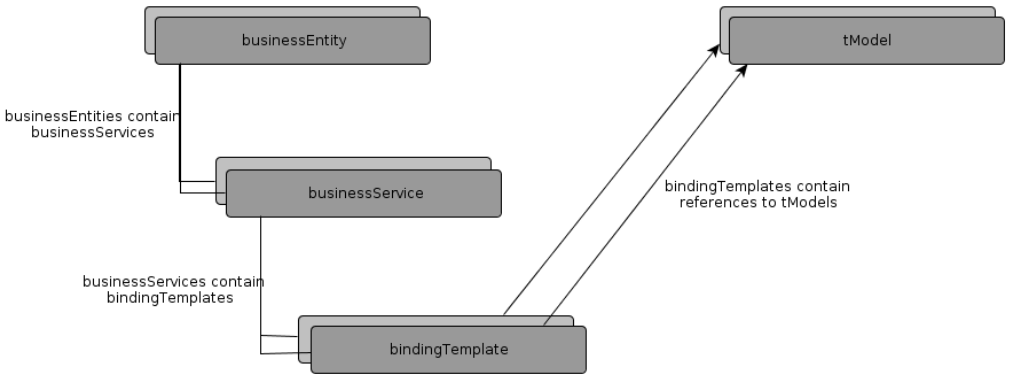


Figure 3.3: UDDI data model

3.2.4 Extensions

The technology stack presented in this chapter provides means for describing, discovering, exposing and consuming Web services. These protocols can be used to cover basic use cases when it comes to invoking distributed application components over the internet. But what if an application needs to provide more complicated features or have a better support for measurable qualities, such as security? To tackle these questions, W3C and OASIS have released a set of specifications that define how topics such as transactions and security should be handled when implementing Web services. As opposed to the first generation label that characterizes SOAP, WSDL and UDDI, these extensions are generally known as the second generation of Web service standards. A subset of these extensions and the functionalities they provide are listed in table 3.2.[19]

Table 3.2: WS-* extensions

Extension	Purpose
WS-Addressing	Provides means to identify Web service endpoints and secure end-to-end endpoint identification in messages.
WS-Policy	Provides means to describe service's requirements and capabilities.
WS-ReliableMessaging	Allows SOAP messages to be delivered reliably between nodes.
WS-Security	Provides quality of protection through message integrity, message confidentiality, and single message authentication.
WS-MetadataExchange	Provides mechanisms to retrieve metadata about a Web service endpoint.
WS-AtomicTransaction	Defines protocols for ensuring automatic activation, registration propagation and atomic termination of Web services.
WS-BusinessActivity	Provides means for systems to interoperate across trust boundaries and vendor implementations.
WS-Coordination	Provides protocols that coordinates actions of distributed applications.
WS-Notification	Provides a protocol for a notification-based interaction pattern.
Web Services Resource Transfer	Provides means for resource access and manipulation.
Web Services Resource Framework	Provides a set of operations that Web services can implement in order to become stateful.
WS-Enumeration	Enables enumerating object instances based on a given filter.
Web Services for Remote Portlets	Enables communication between remote portlets.
WS-Agreement	Specifies an agreement between two parties, such as a service provider and consumer.
WS-Unified Management	Provides means for handling distributed resources between services.

3.2.5 Applying to Digitraffic

3.2.5.1 Advantages

As the previous sections in this chapter have proven, the Web service technology stack standardized by W3C and OASIS is quite complex and the entry barrier for building a system with these technologies might get pretty high. Nonetheless, these standards provide many features that come in handy when designing and implementing a distributed system where the components communicate with each other over the internet. The remaining question is that does the advantage gained from these features exceed the complexity of the framework when regarding mobile client devices and their requirements.

A centric advantage of using SOAP pointed out by Pautasso et al.[36] is that it is transparent and independent when it comes to transmission protocol. In other words, a SOAP message can be transported over multiple

middleware systems, which may rely on HTTP or other protocols. Also quality of service (QoS) aspects, such as encryption and reliable transfer, can be declared in SOAP headers so it does not matter if the carrier protocol changes on the way. However, despite the fact that these features are desired in any sort of distributed application, they are not exactly the main target when looking for a mobile client friendly solution for the Digitraffic API. Other valuable advantages are reliability, security, transactions and service composition[36] that can be achieved by using specific WS-* extensions. The main concern with the use of extensions is that the collection of different extensions is extremely vast and some of the extensions have overlapping features. Therefore, all the service providers and consumers must be using the same extension to achieve a certain functionality in order to co-operate smoothly. As for security and authentication in mobile friendly Web services, many APIs that are consumed with mobile applications use technologies like OAuth[4] to ensure security without using heavy enterprise standards.

Perhaps the main advantage gained by using WSDL is machine-processable definitions of remote requests and responses[36]. This means that errors due to invalid message formats can be caught in compile time and they do not need to be tested individually. From the point of view of mobile mashup applications, this would help the most applications that are developed with static programming languages. For example in browser development this would not help but this would be an advantage when developing native applications and server-side mashups where a static language, like Java[5], is used.

Also with WSDL, a single abstract interface can be tied to different transmission protocols, thus making it more reusable. Pautasso et al.[36] also remind that in addition to communication protocol, WSDL also helps to abstract the underlying serialization details, operating system and programming language. On the other hand, despite the fact that a wide range of tools that can be used to hide the complexity of WSDL from the system developer[36], Richardson and Ruby[39] point out that different tools often produce slightly different WSDL files. Therefore, the client is commonly tied to the same technology stack as the server, thus leaving the components highly coupled.

Even though SOAP and WSDL do not seem like the perfect technology choice for the Digitraffic API, what geospatial services could use is the service discovery provided by UDDI. A properly working service discovery system could help Web service consumption regardless of the platform or architec-

[4]http://oauth.net/2

[5]http://www.oracle.com/us/technologies/java/overview/index.html

tural style. A machine processable, service semantics providing API could remove the manual work now being required to find a Web service to satisfy service consumers' needs. This means that client developers would not have to manually browse through Web sites like Programmableweb[6] to find APIs to consume but some sort of automatic process could be used. Descriptions of service interface semantics could in Digitraffic's case help for example with distance units and velocity measurements (freeflow, average, median etc.). If the semantics of the Web service were clear to the service consumers, there would be no misconceptions if different services are using miles or kilometers or if a velocity is announced as an average or a median value.

The best use case for technologies like SOAP, WSDL and UDDI seems to be found from the enterprise ecosystem where more strict quality of service requirements commonly occur. Since WSDL can model service interfaces for synchronous and asynchronous interaction patterns, it can be used to build gateways for different kinds of legacy systems. The multiple additional features provided by the WS-* extensions allow them to work well in enterprise systems that commonly are built on a varying heterogenous technology environment. All in all, despite their complexity, SOAP and WSDL have gained popularity as gateway technologies providing interoperability of middleware systems.[36] However, many of the enterprise qualities of the WS stack are not the ones that are desired for an API that is consumed by mobile applications.

3.2.5.2 Disadvantages

As mentioned before, technologies like SOAP and WSDL have gained a strong foothold in enterprise system implementations. They have managed to do so due their comprehensive feature set provided by the WS-* extensions and the flexibility of the core technologies. Also the complexity if the RPC interaction pattern is better suited for middleware[47] but when it comes to applications that are targeted for the whole Web, these technologies have not been able to meet the necessary requirements.

The main disadvantages of these technologies regarding the mobile clients, and Digitraffic, is the complexity and low scalability of the solutions that they provide. The complexity issue has lead to a situation where developers have to use heavy tools in order to create service descriptions. Different tools produce slightly different WSDL files, so the service consumers are therefore force to use the same tool set as the server side uses.[39] Also the used tools can reveal the implementation technology of the service[36]. Furthermore,

[6]http://www.programmableweb.com

what also adds into the complexity of the technologies, is that every SOAP interface establishes its own communication protocol[47]. The components communicating with each other need to negotiate an interface contract and a data contract whereas for example with a generic RESTful interface the clients only need to understand the data contract[48]. An extra level of abstraction generated by the interface contract increases the complexity of the system and raises the entry barrier for developing client applications against the service.

The RPC-based approach, which is the case with SOAP, usually results in tight coupling between components[47]. This sort of behavior is not desired when building applications for the Web where the client application developers rarely have any knowledge about the implementation of the server side. The specific interfaces may reveal more about the underlying implementation when compared to generic interfaces and therefore a varying implementation behind a specific interface may also require changes to client applications. The changes to clients can become very expensive to implement[48] and as the number of client applications grow, they can affect the popularity of the API.

The three technologies in this chapter are tied to using XML for markup. The downside with this constraint is the verbosity of XML which results in large messages that require a lot of bandwidth and are slow to process when compared to lighter solutions like JSON[51]. Also it should be noted that an XML description might not match seamlessly with the nowadays popular object-oriented programming languages and that the expressiveness of XML Schema can make it hard to model messages and interface structures in a way that is fully supported by SOAP or WSDL[36].

At this point it seems that the most prominent technology presented in this chapter is UDDI. A repository providing service semantics that could be used to query a correct Web service to be consumed would be useful. However, regardless of its maturity as a technology, UDDI has failed to gain widespread adoption in any sort of environment[36] and it is generally regarded only as part of the first generation of Web service standards[19], whereas SOAP and WSDL are still being used along the second generation known as the WS-* extensions.

To sum up the downsides of the technology stack presented in this chapter, the main problem seems to be that they try to work against the grain of the Web. They are not addressable, well connected of cacheable[39]. The specific interface model adds to the complexity of the services which then again guide the development into direction where heavy tools are required and the entry barrier for client development is high. While HTTP would provide a communication protocol, SOAP builds a new protocol on top of

that and uses HTTP as a mere tunneling protocol[36]. When considering mobile client applications, the biggest disadvantages are heavy tooling and libraries used for development, verbose communication protocol and message format dependent to XML.

3.3 Feeds

Web feeds are data formats that can be used to publish information. Sites can publish their content, or part of it, as feeds that people can access by using a feed reader application. Feeds make it easier for people to find the latest and most significant piece information in the vast amount of content available in the Web. The reader applications can be used to aggregate multiple feeds as an easily accessible way. Web feeds are a popular way to follow news and blogs which usually have frequently updated content that can be easily followed, by subscribing to a feed provided by the news or blog site.

As discussed in chapter 2.1, Web feeds can also be used to gather content into a mashup application. In mashups the feed is parsed in the application and the received information can be presented to the end-user. In this chapter I present two commonly used feed formats, RSS and Atom, and discuss how they could be used to develop a new Digitraffic interface.

3.3.1 RSS

RSS (Really Simple Syndication) is a lightweight XML format that is used for Web feeds. It has a diverse and somewhat confusing history. RSS was originally developed by Netscape for their My.Netscape.com portal in 1999. Back then the version number was 0.9[7] and the acronym stood for RDF Site Summary, RDF (Resource Description Framework) being an XML-based language for describing Web resources and their semantics, developed by W3C. The RDF-based syntax was soon found to be too complicated and the RDF parts were dropped in favor of a cleaner syntax for version 0.91[8]. At this point the name was changed to Rich Site Summary. The development of the RDF-based RSS has led to version 1.0[8] which is currently maintained by W3C. Development of the RFD-less RSS is currently going on in version 2.0[9]. 2.0 is also the version that will be covered in this theses as, according

[7]http://www.rssboard.org/rss-0-9-0
[8]http://www.rssboard.org/rss-0-9-1-netscape
[9]http://www.rssboard.org/rss-specification

to Syndic8[10], it is by far the most widely adopted version.[21][45]

The element structure of an RSS document is presented in figure 3.4.

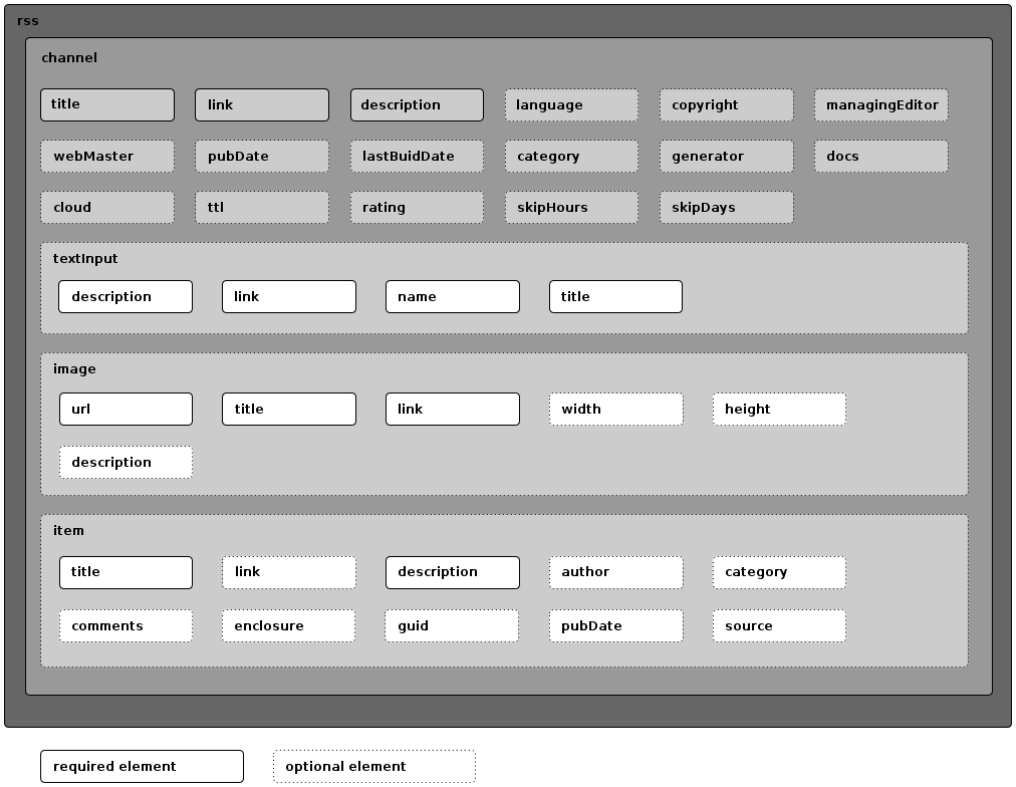


Figure 3.4: RSS structure

The root element in RSS is `rss`. It contains a single sub-element, `channel`, which contains more sub-elements that describe the content and purpose of the document. The most essential elements are listed in table 3.3.

[10]www.syndic8.com

Table 3.3: RSS elements

Element	**Purpose**
channel/title	Name of the channel
channel/link	URL of the corresponding Web site
channel/description	Description of the channel
channel/language	Language of the channel
channel/copyright	Copyright notice for the material published in the channel
channel/pubDate	Date when the channel has been last updated
channel/generator	Implies what application has been used to create the feed
channle/ttl	Indicates the time that the channel can be cached before it has to be refreshed
channel/textInput	Specifies a text input area that can be contained in the channel
channel/textInput/title	Label on the Submit button in the text input area
channel/textInput/description	Describes the purpose of the text input area
channel/textInput/name	Name of the text object in the input area
channel/image	Specifies an image file that can be contained in the channel
channel/image/url	Resource locator of the image
channel/image/title	Title of the image, can be used as the alt attribute of and `<img>` tag if the channel is rendered in HTML
channel/item	An item in the feed, for example a blog post or a piece of news in a newspaper site
channel/item/title	Title of the item
channel/item/link	URL of the item
channel/item/description	Actual content of the item

3.3.2 Atom

Just like RSS, Atom Syndication Format[34] is an XML format that is used to publish information in the Web. It's development was originally started by Sam Ruby of the IBM Emerging Technologies group in June 2003[45] with the original goal to be completely vendor neutral, implemented by everybody, freely extensible by everybody and properly specified[41].

The structure of an Atom feed is presented in figure 3.5.

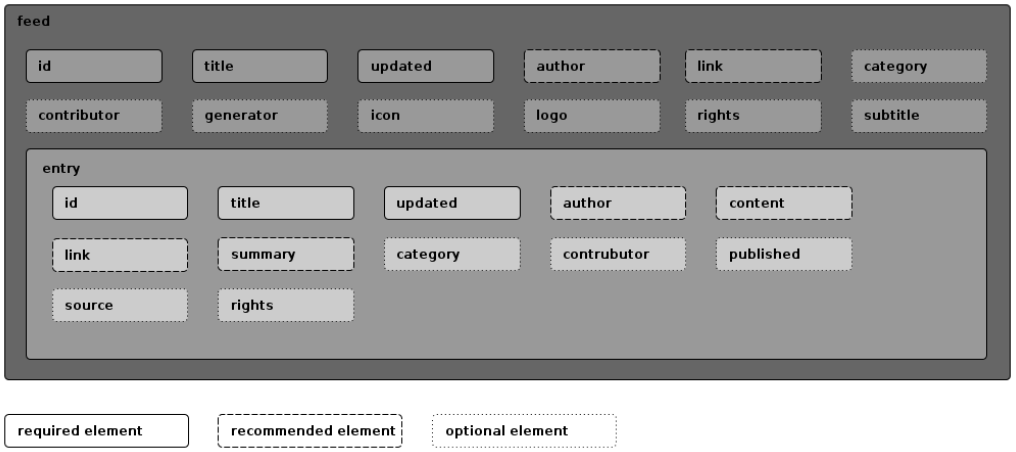


Figure 3.5: Atom structure

Atom has solved some XML encoding issues encountered with RSS. Especially the extensible `content` element enables the inclusion of different kinds of XML content into feed entries. Another difference when compared to RSS is the higher number of required elements. Furthermore, Atom requires an unique identifier for each entry, which prevents clients from listing duplicates when entries are updated.[45] A list of required and recommended elements is presented in table 3.4.

Table 3.4: Atom elements

Element	**Purpose**
feed/id	Permanent URI that identifies the feed
feed/title	Human readable title for the feed
feed/updated	Date when the feed is last updated
feed/author	Name of the author of the feed
feed/link	Link to a related Web page
feed/entry	Single entry in the feed
feed/entry/id	Permanent URI that identifies the entry
feed/entry/title	Human readable title for the entry
feed/entry/updated	Date when the entry was last modified in a significant way
feed/entry/author	Name of the author of the entry
feed/entry/content	Actual content of the entry
feed/entry/link	Link to a related Web page
feed/entry/summary	Short summary of the entry

3.3.3 Applying to Digitraffic

3.3.3.1 Advantages

Evaluating the applicability of feeds to Digitraffic is somewhat challenging task, when comparing to other design paradigms and technologies that are present in this thesis. The amount of research about using feeds in this sort of system design is much smaller than for example with REST or the standardized WS stack. However, the features of feeds and how they are generally used make them a serious option when figuring out the technology that is best suited in this case. After all, feeds are usually used to retrieve information that is regularly updated, like news for example. In the same way they could be used to retrieve traffic information which is updated on regular basis.

One thing that promotes the use of feeds for geospatial systems is the research by Liu and Wilde[30] on *Tiled Feed Model*, which describes how location data can be accessed using feeds. Liu and Wilde use an Atom feed to model tiles that can be divided into smaller tiles. Each tile is published as a single feed with simple spatial extensions. For the client implementations, they have used a standard Atom reader component, which reduces the loose coupling between the service and the client. Some overlapping with competing paradigms might also exist because Liu and Wilde describe Atom as a "RESTful access to collections of resources". However, in this thesis feeds and RESTful architectures are divided into their own paradigms.

3.3.3.2 Disadvantages

The general disadvantages of feeds as an integration tool are pretty much the same as with any other lightweight technology or design paradigm. The lack of features like end-to-end security, transactions, reliability, service composition and service discovery is present also in the case of feeds. However, in the context of Digitraffic, none of these qualities are the main objective. What is is desired is a technology that provides a lightweight access to data on the Web.

A more serious issue with feeds is that they always provide data in XML format. As mentioned earlier in this thesis, more lightweight formats, like JSON, are faster to process which is an advantage when regarding performance[36][51]. Less verbose data formats also optimize the use of bandwidth, which is critical in a mobile environment.

3.4 Other Possibilities

Here I list some technologies that could be used to solve the problem of this thesis but that do not appear to be promising enough for a more detailed study.

3.4.1 XML-RPC and JSON-RPC

XML-RPC[52] is a remote procedure call protocol which tries to enable the use of complex structures but still maintain simplicity of use. It uses HTTP POST to send XML encoded messages between client and server. The messages have a simple structure that describes the method being called and the parameters given to the method.

Similiar to XML-RPC is JSON-RPC[3] which, as the name suggests, uses JSON to serialize the procedure calls. Otherwise it is much like XML-RPC with a simple structure that can be used to model service requests and responses.

These two RPC-style protocols could provide an interface solution much like REST for Digitraffic. The biggest difference is that they do not promote general interfaces as the message is always carried in HTTP POST payload. This adds the interface contract on top of the data contract in the client server interaction. Also there would be some overhead in message sizes compared to REST as the request and response message structure is defined by both protocols. It is to be noted, that even though neither protocol is independent of the encoding format, a XML and JSON interface could be provided by implementing the interface with both protocols. However, this is not as smart way of providing support for different formats for example compared to the RESTful way where the format of the response is announced by the client in the request.

3.4.2 Twitter

Twitter[11] is a social microblogging service that allows users to publish short update status messages, i.e. *tweets*, and subscribe to status updates published by other users. With its current 140 million active users and 340 tweets per day[5] it has an architecture that scales for large queries and massive data traffic.

Twitter also provides a RESTful API[12] that can be used to post and

[11] www.twitter.com

[12] https://dev.twitter.com

query tweets. This could be applied to Digitraffic by making each road link a Twitter user and the last status update of that user could include the current traffic situation on that the link represented by the Twitter account.

The downside of this solution is that Twitter limits status update to 140 characters as it wast originally designed to be SMS-based. Therefore, the polyline data should be retrieved elsewhere and the tweets would only include the traffic fluency information.

3.4.3 Custom Protocol

One possibility could be to design a custom protocol. One way to do this would be to design a new message protocol that is optimized for traffic data. This would include designing some sort of interface contract that the client and server would apply to and the message structure that can be used to pass information between the system components. The emphasis should be in serializing traffic data.

Another possibility is to design the protocol straight to the application layer. One way to do this is to use Blocks Extensible Exchange Protocol (BEEP)[40]. It is a network application framework that enables designing application layer protocols that can be used in network based systems. In other words it can be used to design alternatives to HTTP and other application layer protocols. BEEP could be used, for example, to create a carrier protocol for SOAP. It could also be used to design a network protocol that applies to the constraints of REST. Anyhow, designing a custom protocol for the application layer could enable efficiency that can not be achieved by designing the protocol on top of the application layer.

From the mobile point of view, a custom protocol could be used to achieve more optimized request and response transmission. When considering Digitraffic, the serious downside would be that it would be really hard to gain widespread adoption among client application developers. The new protocol should first establish itself as a known solution for transmitting traffic data and maybe then it could be used to provide Digitraffic data to third parties.

3.5 Solution Comparison

In table 3.5 I have listed the technologies presented in this chapter. They are evaluated against the requirements presented in chapter 1.2.

Table 3.5: Solution comparison

	Mobile friendly	**Platform Independent**	**Format Indepenent**	**Location-Aware**
REST	x	x	x	x
WS Stack				x
Feeds	x	x		x
XML-RPC & JSON-RPC	x	x		x
Twitter	x	x		
Custom Protocol	x	x	x	x

Table 3.5 presents the features of each presented solution. The mobile friendliness measures how well a technology can be used in mobile development environment. REST, XML-RPC, JSON-RPC and Twitter only require that the platform is able to perform HTTP calls. Feeds are also considered mobile friendly as feed readers are extensively used in mobile phones. The WS stack requires heavy libraries in order to work and is therefore not considered mobile friendly.

Platform independence goes hand in hand with mobile friendliness. HTTP calls can be made from any mobile device with an internet connection. WS stack is mainly targeted for enterprise integration and is therefore not optimized, for example, for browser-based client applications.

REST does not set any limits for the data format used in the client-server interaction. Other options are limited to XML (WS stack, feeds and XML-RPC) or JSON (JSON-RPC, Twitter).

All the technologies can be used to design an API that is location-aware as this is a question of parameters that the API takes. Twitter also supports location information in tweets but not in a way intended by a traffic monitoring system.

Custom protocol was left out of the above analysis even though it has all the qualities that are desired from the new API solution. However, creating a custom protocol would significantly harm the adoption of the traffic service API.

Based on this comparison, the new Digitraffic API will be designed by following the REST design principles. The generic interfaces and resource oriented design paradigm can be used to model an API that provides traffic fluency information. In order to minimize message sizes, JSON will be used

to serialize server responses. It is also well supported by client application technologies.

Chapter 4

Implementation

In this chapter I describe the practical part of this thesis. Based on the technological studies of chapter 3 I implemented a new remote API for Digitraffic and a client applications that consumes the API. First I describe how the API was developed and what kind of problems did I have to overcome during the process. Then I describe the client application development process and what kind of new API requirements did it induce.

4.1 New Traffic Data API

4.1.1 Digitraffic Architecture And Technologies

Digitraffic consists of five main components:

CORE Includes data access objects, database queries and data processing.

WEB Web and SOAP interfaces.

DAEMON Polls traffic fluency data form Traffic Data Center to the Digitraffic database.

DATABASE Contains traffic fluency data for each link, does not hold spatial data.

MAP SERVER Contains geospatial information about the links. In other words, knows the shape and location of the links.

The system is written in Java[1] on the Spring framework[2]. The map server is an Esri[3] ArcGIS server. It holds all the spatial information in the system whereas the actual Digitraffic takes care of the traffic data.

4.1.2 RESTful Interface

The new RESTful interface was added to the Web layer in the system architecture. I chose to use the Jersey[4] framework to for implementation as it was easy to add to the existing Spring application.

Using the Jersey implementation, I exposed two resources that provide representations to links and their traffic fluency data. The resources are described in chapter 4.1.6. Digitraffic Web interface obviously has user authentication but I disabled the authentication for the RESTful interface because the authentication methods of a RESTful interface are out of the scope of this thesis.

4.1.3 Spatial Query

Adding a RESTful interface to Digitraffic was relatively easy but working with the map server turned out to be full of different obstacles. My original intention was to retrieve the links the same way that the Web interface does. This turned out to be problematic as the Web interface uses ArcGIS JSF[5] components that can be passed on to the map server. This was not possible with the RESTful interface. Also the existing SOAP interface gave no clues because it creates its traffic reports using batch processes, not dynamic queries. Therefore, I ended up implementing the spatial queries using a low-level ArcObjects API.

Another problem encountered while working with the map server were coordinate systems. The ArcGIS server used by Digitraffic relies on a Finnish coordinate standard called Karttakoordinaattijärjestelmä (KKJ, Map Coordinate System) which is a close relative Universe Traverse Mecator system (UTM). However, most map services, like Google Maps for example, use the World Geodetic System (WGS). Therefore, I had to make coordinate conversions when passing the query parameters to the map server and when receiving the link geometries.

[1] http://www.oracle.com/us/technologies/java/overview/index.html
[2] http://www.springsource.org
[3] http://www.esri.com
[4] http://jersey.java.net
[5] http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

Finally, resolving the points of a polyline object returned by the ArcGIS server turned out to be quite problematic. As an efficient solution was not found and this made the spatial queries really slow, I implemented a cache that holds the polylines (links) in WGS format. This way the polyline is resolved only the first time it is retrieved from the map server. So, when a spatial query is executed, the map server returns a set of polylines that mach the query criteria. Then the cache is checked against the IDs of the polylines and if a polyline is not found form the cache, its points are resolved from map server and saved to the cache. The cache never expires as the roads practically never change. If the cache needs to be flushed, the server can be booted.

4.1.4 Traffic Data Query

When the links that correspond to the given query criteria are fetched from the map server, their traffic fluency information has to be fetched from the Digitraffic database. This was done by using Digitraffic core components.

4.1.5 Response Generation

To form a JSON representation I designed a traffic data Java object that holds the information required by the Web service invocation. To serialize this object to JSON I used the Gson framework[6].

4.1.6 API Description

The new Digitraffic API consists of two resources that are described here. The first one represents traffic fluency.

GET /traffic/fluency

This is the main traffic fluency method. It takes the `latMin`, `lngMin`, `latMax` and `lngMax` parameters that define an extent which narrows down the area where the returned road links are located. In other words, those coordinates define a spatial query that can be used to reduce the retrieved data to apply to the road links that are located near to the user. The `includePolylines` can be set to `true` or `false`. By changing this value the client can control if only the fluency data is returned from the server or if

[6]http://code.google.com/p/google-gson

also the actual polylines are also included in the response. Parameters and their purposes are listed in table 4.1.

Table 4.1: Fluency method parameters

Parameter	**Purpose**
latMin	Minimum latitude value
lngMin	Minimum longitude value
latMax	Maximum latitude value
lngMax	Maximum longitude value
includePolylines	Indicates if the actual polylines should be included in the response

The method returns a list of JSON objects that contain a polyline, basic info and the latest traffic fluency data. An example JSON snippet is displayed in Example 4.1.

```
[
  {
  "polyline":
      {
      "linkId":24,
      "points":
          [
              [24.82207,60.18085],
              [24.82104,60.18126],
              ...
              [24.81936,60.21171],
              [24.81939,60.21195]
          ]
      },
  "linkId":24,
  "length":3830,
  "name":"Otaniemi-Perkkaa",
  "averageSpeed":46.424,
  "medianTravelTime":297,
  "fluencyClass":
      {
      "lowerLimit":0.75,
      "upperLimit":0.9,
      "code":4,
      "name":"Liikenne jonoutunut"
      }
  },
```

```
  ...
]
```

Example 4.1: Fluency resource response

The example snippet presents a list of traffic fluency objects (here only the first of the list is shown) with the link ID, the points that build up the polyline, basic link info and the fluency class object which then again contains info about the current fluency class on that link. The fluency class is an integer between one and five and it indicates the fluency of the traffic on a link. This kind of response can be received by setting the `includePolylines` parameter to `true`. Otherwise the response will be exactly the same but the `polyline` field will be left out of the JSON objects.

The other method in the new API is as follows:

GET /traffic/link/{link id(s)}

This method returns the polylines that correspond to the given link IDs. Multiple IDs can be given in the URL by delimiting the with a comma, for example `http://www.digitraffic.com/traffic/link/101,102,103`. An example response message is presented in example 4.2.

```
[
  {
    "linkId":8,
    "points":[
      [24.85097,60.23037],
      [24.85196,60.23057],
      ...
      [24.93048,60.24083],
      [24.93108,60.2409]
    ]
  },

  ...

]
```

Example 4.2: Link resource response

The division between these two methods allows clients to use different kinds of caching schemes with the road links and their fluency data. For example, a client can always retrieve the polyline data when fetching the fluency information by setting `includePolylines` parameter to `true` in the first method. This ways the client always gets the polyline data with each

request and therefore this kind of fluency data handling is easy and simple to develop. In order to use more efficient caching, client applications can leave the polylines out of the fluency request and fetch polylines separately with the `link` resource identifier. As this resource can be cached, clients can cache the polylines and only retrieve new polylines that have not been retrieved previously. HTTP header fields of the `/traffic/link` resource are presented in example 4.3 and as can be observed, the `max-age` value is given in the `Cache-Control` field.

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=85CBF039280F837B2B42DA2F77B1AFBA; Path
   =/sujuvuus
Cache-Control: no-transform, max-age=2592000
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 25 Apr 2012 14:47:13 GMT
```

Example 4.3: Link resource response HTTP header

Both resources also support gzip compressed response messages. The compressed responses can be fetched by setting the `Accept-Encoding` HTTP request header field to `gzip`. A HTTP header of a compressed response message can be seen in example 4.4.

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=CB68500BE35886EF830B665BD90AB094; Path
   =/sujuvuus
Content-Encoding: gzip
Content-Type: application/json;charset=UTF-8
Content-Length: 20
Date: Mon, 14 May 2012 04:55:55 GMT
```

Example 4.4: Compressed response HTTP header

4.1.7 Lessons Learned

Releasing a RESTful API usign the Jersey framework was a straightforward process. By using annotations and few configuration changes, an interface was added to the Digitraffic Web layer in no time. The most time consuming part of the server side implementation was integrating the REST API with the ArcGIS server that holds the link geometries and their locations.

One problem with the ArcGIS server was the coordinate system mismatch. The number of standards is huge and there really is work to be done to unify different systems and the coordinates they use. Luckily the WGS sees to be getting a foothold in application development as most map services use it. I strongly suggest that geospatial systems developed in the future use WGS coordinates. However, I don't think that migrating Digitraffic to WGS is worth the effort as too many things might break.

Another problem with the ArcGIS integration was the complexity of the ArcObjects API that was used to access the polyline information. The API is hard to use and its documentation is hard to read. One real annoying feature are the error stack traces which give no clue of the problem whatsoever. Implementing spatial queries form scratch was the most time consuming part of the server-side development. Future work should try to take advantage of libraries, like the JSF-based library used by the Digitraffic Web interface, that ease the ArcGIS server integration work.

One unsolved problem with the ArcGIS server API was resolving the points of a polyline. The only way I was able to do this was loop through all the points in a polyline and request point coordinates from the ArcGIS server one by one. As mentioned in chapter 4.1.3, this problem was solved by caching the polylines. In future development a better solution for this should be found.

4.2 Client Application

As part of the thesis I also designed and implemented a client application that consumes the new Digitraffic API. The main idea was to concentrate on the design and implementation of a new remote API but the client also serves a purpose as it can be used to prove that the new API actually works on practical level in a real life use case

The implemented client is a mobile location-aware map application that visualizes the traffic fluency on roads located close to the user. It locates the user and then uses bounding coordinates of the map to send an optimized traffic fluency data query to Digitraffic.

4.2.1 Mobile Platforms

One main task when developing the client side was to choose the mobile platform the application was to be built on. Currently native application platforms compete with mobile browsers for application share. Both solutions provide advantages and disadvantages from the point of view of developers

and end users. Here present different options that are available when it comes to mobile application platforms and justify the decision I made with the mobile application platform for the Digitraffic client.

4.2.1.1 Mobile Operating Systems

The two most prominent mobile operating systems I considered to be used for the client are Apple's iOS[7] and Google's Android[8]. Using native platforms for mobile application development has it's up- and downsides. Native applications run efficiently and can easily use all features provided by the platform like camera and accelerometer for example.

The downside with native applications is that every application has to be implemented separately for each and every platform. This is time consuming and requires knowledge of all the platforms that an application is intended to run on.

4.2.1.2 HTML5

An interesting new mobile platform is the Web browser ie. HTML5[9]. HTML5 is the newest version of the Web markup language but is usually used as an umbrella term to cover new versions of technologies like HTML, CSS and JavaScript. The advantage in developing browser based applications is that a single application can be run in any modern smartphone. Browsers in new mobile devices usually have a good support for HTML5-related technologies so the problem of supporting older browsers is not such an issue as it is with applications intended for desktop browsers. It is a lot easier to gain good skills in Web application development than to learn to develop native applications for every major mobile platform. This makes the browser a considerable mobile platform choice.

The downside of browser based applications is their lower performance when compared to native applications. They also can not use all phone features like native applications can.

4.2.1.3 PhoneGap

To combine the good sides of native mobile applications and the ease of development of mobile Web applications, I chose PhoneGap[10] as the platform

[7] www.apple.com/ios
[8] www.android.com
[9] www.w3.org/html/wg
[10] www.phonegap.com

for the Digitraffic client application. PhoneGap is a mobile application development framework that allows applications to be developed using Web standards and to be deployed into phones as native applications. To achieve this, PhoneGap provides a JavaScript interface that can be used to access the native features of a device. An application developed with browser technologies can then be packaged and installed as a native application on any major mobile platform, in my case Android.

4.2.2 Development Tools

As mentioned, the client application was developed on top of PhoneGap. The target phone where the application was to be deployed was a HTC Desire Android phone. So the development environment was an Eclipse IDE bundled with Android SDK.

Biggest concern when developing the application was to pick up the best tools and make the work together. Table 4.2 lists the used JavaScript libraries and their purposes.

Table 4.2: JavaScript libraries

Library	**Purpose**
PhoneGap	Phonegap JavaScript library that provides the use of the phones native features and enables the PhoneGap development all together.
jQuery	JavaScript library that simplifies Web development
jQuery Mobile	A JavaScript library optimized for mobile applications. Also provides templates for the user interface
Google Maps	Provides ways to load and handle a Google map component
jquery-ui-map	Eases the use of Google Maps with jQuery Mobile
jquery.toastmessage.js	Used for visual notifications

4.2.3 Application Description

The client application is quite simple. When it launches, the location of the device is determined. The map is centered at this position and a query is sent to the Digitraffic server, giving the coordinates of the corners of the map view as parameters. The returned traffic fluency data is then visualized on the map by drawing the links on it. Colors of the links depend on the fluency class of each road link. Red indicates that the traffic is jammed whereas green indicates fluent traffic. Orange, blue and yellow colors indicate the fluency classes in between. The traffic data on the display can be updated by pressing *Refresh* button.

The visualized road links can be seen in figure 4.1.

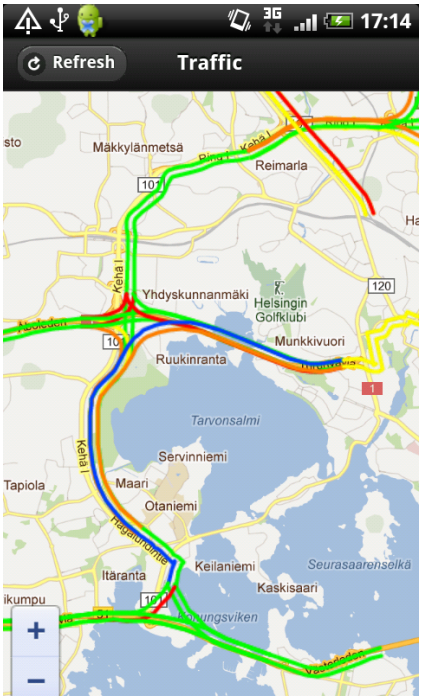


Figure 4.1: Mobile client application

4.2.4 Lessons Learned

Even though the application was deployed as native Android application, it was still developed using Web technologies. While programming the AJAX calls by using the jQuery AJAX API, I started to think about the cross-domain AJAX calls that previously have been problematic to perform.

The cross-domain AJAX call is a problem with all kinds of Web-based mashup applications. The problem is that due to the same-origin policy[7], Web sites can only fetch content from the domain where they are hosted. Web browsers implement the same-origin policy to prevent a security defect called *cross-site scripting*. The problem is modeled in figure 4.2.

Figure 4.2: Problem of cross-domain AJAX requests

In the figure the mobile phone represents a mobile Web application (`mydomain.com/app.html`) that is hosted at `mydomain.com`. The Web site is allowed to make AJAX calls to the `mydomain.com` server but not other servers, like the `digitraffic.com` presented here. One way to overcome this problem is to change the application design into a server-side mashup (described in chapter 2.1). This means that the call to the other domain is performed in the server. However, this solution has its limitations. Calling other services in the server-side increases the workload of the back-end. Furthermore, some open APIs have limitations for the number of calls made from same IP address in a given time. If `digitraffic.com` had this kind of a limitation, the limit could easily be exceeded as all the API calls required by `mydomain.com/app.html` would come form the same IP address. If the `digitraffic.com` calls could be made straight from the Web site, the origin IP would be the client device IP.

Even though the reference client is a native application, it would be desired that also mobile Web clients could use the new API. This enables the use of the API from mobile browsers which seem to be becoming a more and more popular application platform for mobile applications. I mentioned in chapter 2 that HTML5 enables cross-domain AJAX calls[6], but browser support for the cross-site calls provided by HTML5 is varying so now I started to look into different ways to enable them in the new Digitraffic API.

While looking into different options to enable cross-domain calls, I came across a technique called JSONP[11]. The acronym originates from "JSON with padding" and the idea is to exploit the `<scritp>` HTML tag by giving a *callback* function name as an extra parameter to the Web service. The Web service then wraps the response JSON into a function of this name and the response can be retrieved as it was dynamic JavaScript that was added to that page.

To enable JSONP for the new Digitraffic API, I implemented a callback method check to each resource. If a callback parameter is given with an API

[11]http://www.json-p.org

call, the response is wrapped in a method of the name defined by the callback parameter.

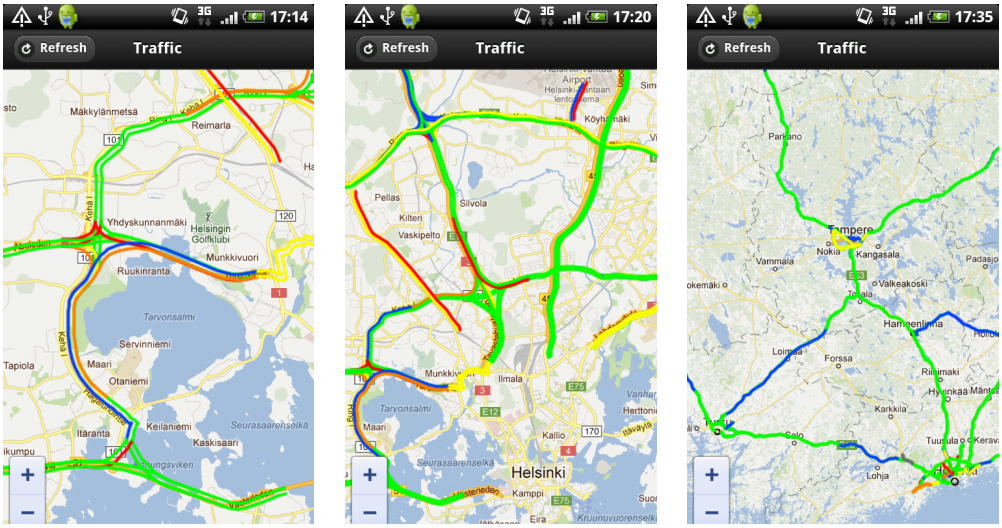
Chapter 5

Testing

In this chapter I present and analyze the results that were gained by testing the new Digitraffic API. I also describe the testing environment that was used to perform the tests.

5.1 Test Environment

The new Digitraffic API was tested by retrieving three different resources. Each resource was retrieved 10 times and the invocations were made with the client application. The three request sets differed in a way that in each set, the map zoom level was different. Different zoom levels are presented in figure 5.1. In each test case traffic data was retrieved for the road links that are visible in the map. Therefore, by decreasing the zoom level, a bigger area on the map was exposed and therefore the traffic data was retrieved for a bigger number of links.



(a) (b) (c)

Figure 5.1: Zoom levels

In case *a* the zoom level is relatively high and only parts of some local highways around Helsinki area can be seen. In case *b* the zoom level is decreased and most roads around Helsinki area are presented in the map. In *c* the zoom level is decreased even more and three major cities, Helsinki, Tampere and Turku are present in the map. These three different cases are used to demonstrate how the location-awareness affects API response message sizes and furthermore, how the message size affects the time used to parse the results in the client device.

Each request set consisted of ten API calls. The retrieved resource was `/traffic/fluency` and the `includePolylines` parameter was set to `true` so every polyline was also returned with each API call. As a reminder, polylines are geometries that define the road links that are under traffic surveillance. Knowing the link geometries enables the visualization of the traffic fluency in client applications. The client application was run in a HTC Desire phone. Test results were collected from the phone and by emulating the client calls to the server from command line. For example, processing times were measured by logging the time it takes to process the responses in the phone but the message sizes were measured by logging the query parameters at the server side and then emulating the same calls with `curl` command on command line. Since the query parameter coordinates were the same as with the calls from the client application, the responses were exactly the same.

5.2 Test Results

Table 5.1 presents the results of the API calls that correspond to the three different zoom levels. The number of polylines found by Digitraffic, message size the API call induces and processing time were measured. In the case of the number of found polylines and response message size, the values were the same on each API call within a request set as the query extent sent by the client application to the API was always the same between calls of a single zoom level. With processing time slight variance was naturally encountered and therefore, a median time is used.

Table 5.1: Test results

Zoom level	**Links**	**Message size**	**Processing time (median)**
(a)	56	77KB	555 ms
(b)	128	220KB	1245 ms
(c)	413	1.2MB	5053 ms

As mentioned, the results in table 5.1 were obtained by fetching the `/traffic/fluency` resource with the `includePolylines` set to `true`. To see the effects of proper caching and response compression, I also performed the test API calls with the `includePolylines` parameter set to `false` and with the response compression enabled. This comparison can be seen in table 5.2

Table 5.2: Message size comparison

Zoom level	**w/ polylines**	**w/ polylines, gzip**	**w/o polylines**	**w/o polylines, gzip**
(a)	77KB	18KB	8.3KB	246B
(b)	220KB	57KB	18KB	2.4KB
(c)	1.2MB	322KB	63KB	7.9KB

5.2.1 Message Size

Response message sizes increase drastically as the zoom level is decreased. The closest zoom level in case *a* clearly produces the smallest response message, as expected. Case *b*, where the main roads in Helsinki area can be seen on the map, produces a slightly larger message of 220 kilobytes. In the case *c* the message size is over one megabyte, which starts to be quite a large response to be received by a mobile device.

This test shows that an interface that enables location-awareness truly can make applications more responsive for the end user as bandwidth is usually

limited when it comes to mobile devices. When the user can be located and the location information can be used to make queries more efficient, interfaces can return data that is more relevant to the user and that data can be consumed in a faster manner by mobile client applications. Case *c* where the traffic situation of three major cities is visualized at once is clearly not an important use case in a mobile traffic application. Case *a* provides a good look on a local traffic situation for a user on the road and the case *b* provides a overview on the traffic on main roads around Helsinki. In cases *a* and *b* the message size stays in tolerable limits.

An other important finding is the effect that caching and response compression have on the response message size. The impact of these two methods is huge and they can effectively be used to increase the mobile-friendliness on an API. As can be seen in table 5.2, by applying polyline caching in the client and response compression in the server the case *a* response message size is reduced to 246 bytes and the case *c* to 7.9 kilobytes. These are significantly smaller messages than the ones that are returned without polyline caching and compression.

To tell what amount of data is too much in mobile environment would of course require the consideration of the bandwidth used. I have not included the transmission times of the messages with different connection bandwidths as the bandwidths are somewhat difficult to explicitly measure. My mobile internet provider Elisa[1] reports that the maximum bandwidth in the 3G network is 15 MB/s which sees like quite a lot when considering how my mobile connection normally works, say for web browsing, with good connectivity. I also tried a mobile connection speed measuring application called Speedtest[2] to test the actual bandwidth of my mobile internet connection. It reported that my bandwidth is slightly higher than 5 MB/s. That is far less than the number announced by my operator but still quite high for a mobile broadband. The true bandwidth also changes depending on how loaded the network is. Also the available bandwidth is reduced in areas where the 3G network is not available and the device has to load the data using slower connections. Due to these reasons I did not include transmission times to the explicit test results table. However, it is safe to say that the message retrieved in cases *a* and *b* can be transmitted in satisfactory amount of time with the mobile internet connections that are currently available.

One observation that led to a change in the API implementation during the tests was the message size and how the precision of the points that compose the polylines affect it. Originally the precision of the coordinates

[1] www.elisa.fi

[2] www.speedtest.net

was 14 fractional digits. This precision was the result of the coordinate conversions after the spatial query was conducted. It meant that most of the content of each response message was composed of the fractional digits. By observing the client application interface and the sharpness of the traffic fluency visualization, I was able to reduce the precision of the coordinates to five digits before any negative effects could be seen in the client application. This procedure reduced the sizes of the response messages roughly to half of the original.

5.2.2 Performance

Processing time gives a view on how much the zoom level affects the user perceived responsiveness after the response message has been received by the client device. The time measured here is how long it takes to parse the response message, create polyline objects that can be passed to the map, resolve the correct color for each polyline according to the traffic fluency class and finally draw the polylines on the map.

The processing time in case *a* is about half a second so it has no drastic effects on the responsiveness of the application. The time it takes to process the response in cases *b* and *c* show that location-awareness and optimized queries are important also when it comes to how fast the retrieved data can be processed in a mobile device.

If compressed responses were used, also the message decompression should be taken into consideration regarding processing times. As the response compression was not implemented in the client side, there are no precise numbers on the decompression performance in the test device. However, the message size reduction accomplished with compressed responses is so significant that it is safe to say that it compensates the lack of performance caused by the decompression.

The performance is highly dependent on the phone where the application is used as the processing power varies between phone models. The phone used for these tests is almost two years old but with its 1 GHz processor it still provides valid results. Also the number of applications that run simultaneously can have an effect on the processing time. Therefore, it is difficult to measure an explicit processing time even though it has to be noted that all the performance tests were run in the same phone so the benefits of location-awareness can be observed.

5.2.3 Client Application Development

One main target in the new Digitraffic API design was to enable simple client applications. This does not mean that the provided traffic data could not be used in complex use cases but that basic applications consuming the data would be simple to implement and would not require heavy enterprise tools. Here I will evaluate the simplicity of the client application implementation.

One way to measure the complexity of client application is simply the number of lines of code used to implement the basic functionalities. As mentioned, the application is implemented using PhoneGap, which means that the implementation consists of HTML, CSS and JavaScript. The main procedures of the Digitraffic client are locating the device, initializing the map, retrieving traffic data from Digitraffic, parsing the data and drawing the links on the map. All this is performed with approximately 160 lines on JavaScript code. Most of the lines are used to initialize the map and resolve the traffic fluency classes of retrieved road links. Parsing the retrieved data requires practically no effort since the interface returns JSON which is automatically parsed into valid JavaScript objects.

The client implementation source code is presented in appendix A.

5.2.4 Comparing to the Old Solution

Comparing the new API to the old one is not trivial. One reason is that the new interface does not correspond to the old interface because, the use case of the new interface is different. However, the old interface provides a method called `trafficFluency` that returns traffic fluency data for each link in the Digitraffic system. The size of an example response from this method is 276 kilobytes. It is to be noted that this response only includes the traffic data but not the link geometry information. As for comparison, retrieving traffic data for every link in Digitraffic and excluding the link geometries from the response by setting the `includePolylines` to `false`, the new RESTful API returns a response of 97 kilobytes. So by using the new API a decrease of 64.9% in message size can be achieved. By using the location based query optimization features provided by the new API, fetching only the fluency data would lead into even smaller responses in real life use cases.

Chapter 6

Discussion

In this chapter I discuss the meaning of the test results and the succeeding of the implementation. I also present possible improvement ideas.

6.1 Reliability of the Test Results

The test results do not provide any drastic surprises. Reducing the zoom level leads to a bigger amount of links to process and this led to bigger response sizes. Some skepticism though has to practiced when analyzing the processing times. The processing time can vary based on the processing power of the phone that is used. Also different background processes running in the phone can affect these values. Anyhow, I believe that ten measurements and the use of median time give a good impression on how well the location context and spatially optimized queries affect the processing time.

The reliability of the results is slightly reduced by the lack of measurements about the transmission times of different message sizes on different bandwidths and the lack of measurements about the gzip decompression performance of the test phone. Especially the decompression measurements would have been a good addition to the test result set.

6.2 Applicability of the Selected Solution

The RESTful approach and the use of the JSON format proved to be a good choice for the new API implementation. The general REST interface allows the development of lightweight client applications that suite form mobile client devices. Using JSON was a good choice regarding the client development as practically no code lines had to be used to parse the responses. JSON format enables the use of JSONP in which the response message is

wrapped as a JavaScript function in the server. It enables cross-domain AJAX calls from Web sites and therefore has a huge positive impact on the client independence of the new API.

6.3 Optimizing the Response Messages

As can be seen in the message size comparison in table 5.2, a proper caching scheme and response compressing can seriously improve the response message size. From the client point of view, implementing the polyline caching would mean using both `/traffic/fluency` and `/traffic/link` resources. The `/traffic/fluency` resource would be retrieved with the `includePolylines` parameter set to `false` and missing link information would be retrieved by fetching the `/traffic/link` resource for those links that are not saved in client application cache. This would require some extra design and implementation work in the client-end. The response compression could be easily added to the client end as it could be handled by a JavaScript framework.

One question about the cache control is how long the link data is valid. Roads are practically static data so the link geometry information can stay valid for years or even longer. Currently the cache control is set to 30 days, but maybe it could be extended from this. Then again, if a road is changed, the changes should be visible in traffic fluency applications instantly. I propose that 30 days is a good validity time for a link geometry.

Another way to decrease the amount of transmitted data on each update would be to use a better server-client interaction model. Currently the API and the client rely on traditional request-response model, where the server sends a response each time the client calls it. The problem with this design is that it does not recognize changes in resource state. When the client retrieves a resource, the resource state might be the same as the last time the client retrieved it. In other words, when the client updates the traffic fluency data, the update call is unnecessary if the traffic status has not changed. One way to tackle this problem would be to use a *push-technology* to initialize the client-server interaction from the server-side whenever there is a change in the traffic situation. One possible push solution could be WebSocket[23] which is part of the HTML5 standards.

6.4 Visual Impacts of the Zoom Level

One observation that can be made for example from figure 5.1 is the way the links are drawn in the current client implementation. When the spatial

query is conducted behind the REST API, an offset is given to the links that go to opposite directions. This way the lanes are not drawn on top of each other. The width of a link is always constant in the client and this leads to a behavior where the lanes are hard to distinguish from each other when the zoom level gets lover. However, in order to fix this, the client has to be improved as the links always have to be spatially same when they are returned from the API.

6.5 Improvement Ideas

One possible improvement could be to provide different kinds of resources with the API. Currently only fluency and link data can be retrieved but Digitraffic holds also other kinds of data like road conditions. Also the traffic fluency data could be provided in other ways like reports of previous traffic information and different kinds of calculations about the data.

In the client side, some major improvements could be conducted by implementing automatic data refreshing. Now the user has to manually refresh the application by pushing the *Refresh* button. A better behavior would be if the traffic situation was automatically updated in a regular interval. The question here is that how often information should be updated i.e. how fast does traffic fluency data become obsolete. On some road blocks it could happen in minutes so an interval of 5-10 minutes would be optimal. Then again, in that time the driver might drive so far from the location where the data was previously updated that the new traffic information is received too late to have an impact on the routes the driver chooses.

Another improvement idea related to this is the tracking of the user. Now the application does not track the user, but the user has to define the location for which the traffic fluency data is fetched, except for when the application is started and the location is resolved by the application. If a person is driving down a road, it would be desired that the map was always focused in the location of the user. Also it would come in handy if the location of the user was pointed out in the map.

GPS navigators and many map services provide route planning which helps the users to decide which is the best route to reach their destinations. Route planning could also bring great additional value to Digitraffic and its mobile client. Unfortunately Digitraffic does not provide this kind of information. So the way to add the planning info to the client would be to create a mashup that retrieves the traffic fluency data from Digitraffic and the route planning information from some other source. However, this would increase the amount of processing in the client when it would combine these

two information sources and calculate the best routes. One solution would be that Digitraffic retrieves the planning data from a third party API, combines it with the traffic data and the provides the traffic-aware route planning through its API. However, this would slightly alter the original purpose of Digitraffic.

Chapter 7

Conclusion

This thesis studied different ways of providing traffic data to location-aware mobile devices. Based on the results I implemented a new remote programming interface to an existing traffic surveillance system.

The technologies and architectural designs I looked into were REST, Web service standard stack, feeds, XML-RPC, JSON-RPC, Twitter and custom protocols. The first three ones were studied in detail because they were the most prominent technological solutions for the problem at hand.

From these possibilities, REST was selected as the approach to be used when developing a new traffic fluency data API. Based on this decision I implemented a new RESTful API that provides JSON representations of traffic fluency and road link resources. The new API provides location-based query optimization and a caching scheme for road link information.

To test the new interface, I implemented a mobile client application. The client was used to show that the new API works and to measure the performance of the API. The test results showed that optimizing traffic data queries improves the responsiveness of a mobile traffic monitoring application.

The practical benefits of this work include the possibility to use the data provided by Digitraffic in mobile phones. The real production version of Digitraffic does not yet include this interface but the results gained with the prototype described in this thesis show that a RESTful API can be used to distribute the traffic data to mobile phones. This thesis also shows that the use of location data can make data queries more efficient and therefore make the system more mobile-friendly.

Based on this thesis work, I propose the following guidelines for a Web service that is intended to be mobile-friendly and take advantage of the location-aware features of client devices.

- Design the interface according to RESTful architectural guidelines. This way the Web service provides a generic interface that can be accessed from any kind of application platform that supports internet connection.

- JSON format can be applied to describe varying data structures. It enables compact messages and is widely supported by client technologies. Using JSON as data format enables the use of JSONP for cross-domain data transfer in browser-based client applications. JSONP is supported by every browser that implements the `<script>` HTML tag so it is more widely adopted than other cross-domain methods.

- Provide response customization based on user location. In order to support map components used by client applications, prefer internationally standardized coordinate systems like WGS rather than national coordinate systems.

- Provide cache-control information for static data. Divide resources so that static and non-static data can be retrieved separately.

- Provide request and response compression in order to optimize message sizes.

These guidelines can be applied in future projects at Gofore. They apply for systems that hold spatial information and where client applications need to be deployed in mobile environment.

Bibliography

[1] ECMAScript Language Specification. , ECMA International, 1999. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[2] Introduction to UDDI: Important Features and Functional Concepts. , OASIS, 2004. http://uddi.org/pubs/uddi-tech-wp.pdf.

[3] Json-rpc 2.0 specification. , JSON-RPC Working Group, 2009. http://www.simple-is-better.org/json-rpc/jsonrpc20.html, Cited: 20.4.2012.

[4] Pääministeri Jyrki Kataisen hallituksen ohjelma. , Valtioneuvoston kanslia, 2011. http://www.vn.fi/hallitus/hallitusohjelma/pdf332889/fi.pdf.

[5] Twitter turns six. , Twitter Blog, 2012. http://blog.twitter.com/2012/03/twitter-turns-six.html, Cited: 25.4.2012.

[6] Aghaee, S., and Pautasso, C. Mashup development with html5. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups* (2010), ACM.

[7] Barth, A. The Web Origin Concept. 6454, IETF, 2011. http://www.ietf.org/rfc/rfc6454.txt.

[8] Bedge-Dov, G., Brickley, D., Dornfest, R., Davis, I., Dodds, L., Eisenzopf, J., Galbraith, D., Guha, R., MacLeod, K., Miller, E., Swartz, A., and van der Vlist, E. Rdf site summary (rss) 1.0. http://web.resource.org/rss/1.0/spec, 2000.

[9] Bellwood, T., Capell, S., Clement, L., Colgrave, J., Dovey, M. J., Feygin, D., Hately, A., Kochman, R., Macias, P., Novotny, M., Paolucci, M., von Riegen, C., Rogers, T., Sycara, K., Wenzel, P., and Wu, Z. Uddi

version 3.0.2. UDDI Spec Technical Committee Draft, OASIS, 2004. http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm.

[10] Berners-Lee, T., Fielding, R., and Masinter, L. Uniform resource identifier (uri): Generic syntax. , Network Working Group, 2005. http://tools.ietf.org/html/rfc3986.

[11] Bray, T. Smex-d (simple message exchange descriptor). http://www.tbray.org/ongoing/When/200x/2005/05/03/SMEX-D, Cited: 15.5.2012, 2005.

[12] Bray, T., Paoli, J., Maler, E., Yergeau, F., and Sperberg-McQueen, C. M. Extensible markup language (XML) 1.0 (fifth edition). W3C Recommendation, W3C, 2008. http://www.w3.org/TR/2008/REC-xml-20081126.

[13] Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. Web Service Definition Language (WSDL). W3C Note, 2001. http://www.w3.org/TR/wsdl.

[14] Chumley, R., Durand, J., Pilz, G., and Rutt, T. Ws-i basic profile version 2.0. , WS-I, 2010. http://www.ws-i.org/Profiles/BasicProfile-2.0-2010-11-09.html.

[15] Cowan, J. Resdel. http://recycledknowledge.blogspot.com/2005/05/resedel.html, Cited: 15.5.2012, 2005.

[16] Curbera, F. b., Duftler, M. b., Khalaf, R. b., Nagy, W., Mukhi, N. b., and Weerawarana, S. b. Unraveling the Web services Web: An introduction to SOAP, WSDL, and UDDI. *IEEE Distributed Systems Online 3*, 4 (2002).

[17] Ennals, R., Brewer, E., Garofalakis, M., Shadle, M., and Gandhi, P. Intel mash maker: join the web. *SIGMOD Rec. 36* (2007).

[18] Erenkrantz, J. R., Gorlick, M., Suryanarayana, G., and Taylor, R. N. From representations to computations: the evolution of web architectures. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (2007), ACM.

[19] Erl, T. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[20] Fielding, R. T. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[21] Garofalakis, J., and Stefanis, V. Using rss feeds for effective mobile web browsing. *Universal Access in the Information Society 6* (2007).

[22] Hadley, M. Web application description language. `http://www.w3.org/Submission/wadl`, Cited: 15.5.2012, 2009.

[23] Hickson, I. The web sockets API. W3C Working Draft, W3C, 2009. `http://www.w3.org/TR/2009/WD-websockets-20091222`.

[24] Joe, Z., and Pavlovski, C. Towards accountable enterprise mashup services. *Proceedings - ICEBE 2007: IEEE International Conference on e-Business Engineering - Workshops: SOAIC 2007; SOSE 2007; SOKM 2007* (2007).

[25] Kopecký, J., Gomadam, K., and Vitvar, T. hrests: An html microformat for describing restful web services. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01* (2008), IEEE Computer Society.

[26] Lanthaler, M., and Gütl, C. Towards a restful service ecosystem: Perspectives and challenges. *4th IEEE International Conference on Digital Ecosystems and Technologies - Conference Proceedings of IEEE-DEST 2010, DEST 2010* (2010).

[27] Li, L., and Wu, C. Design patterns for restful communication web services. *ICWS 2010 - 2010 IEEE 8th International Conference on Web Services* (2010).

[28] Lin, S.-Y., Chao, K.-M., and Lo, C.-C. Service-oriented dynamic data driven application systems to urban traffic management in resource-bounded environment. *SIGAPP Appl. Comput. Rev. 12*, 1 (2012).

[29] Liu, X., Hui, Y., Sun, W., and Liang, H. Towards service composition based on mashup. *2007 IEEE Congress on Services* (2007).

[30] Liu, Y., and Wilde, E. Scalable and mashable location-oriented web services. In *Proceedings of the 10th international conference on Web engineering* (2010), ICWE'10, Springer-Verlag.

[31] Mitra, N., and Lafon, Y. SOAP version 1.2 part 0: Primer (second edition). W3C Recommendation, W3C, 2007. `http://www.w3.org/TR/2007/REC-soap12-part0-20070427`.

[32] Mokbel, M., Aref, W., Hambrusch, S., and Prabhakar, S. Towards scalable location-aware services: Requirements and research issues, 2003.

[33] Murugesan, S. Understanding web 2.0. *IT Professional 9*, 4 (2007).

[34] Nottingham, M., and Sayre, R. The atom syndication format. `http://www.ietf.org/rfc/rfc4287.txt`, 2005.

[35] Orchard, D. Wdl (web description language). `http://www.pacificspirit.com/Authoring/WDL`, Cited: 15.5.2012.

[36] Pautasso, C., Zimmermann, O., and Leymann, F. Restful web services vs. "big"' web services: making the right architectural decision. In *Proceeding of the 17th international conference on World Wide Web* (2008), WWW '08, ACM.

[37] Poikola, A., Kola, P., and Hintikka, K. A. *Public Data - an introduction to opening information resources*. Edita Prima Oy, 2010.

[38] Prescod, P. Web resource description language ("word-dul"). `http://www.prescod.net/rest/wrdl/wrdl.html`, Cited: 15.5.2012.

[39] Richardson, L., and Ruby, S. *Restful web services*, first ed. O'Reilly, 2007.

[40] Rose, M. The blocks extensible exchange protocol core. `http://tools.ietf.org/html/rfc3080`, Cited: 20.4.2012, 2001.

[41] Ruby, S. Roadmap. , The Atom Wiki. `http://www.intertwingly.net/wiki/pie/RoadMap`, Cited: 15.5.2012.

[42] Salminen, A., Kallio, J., and Mikkonen, T. Towards mobile multimedia mashup ecosystem. *IEEE International Conference on Communications* (2011).

[43] Salo, J., Aaltonen, T., and Mikkonen, T. Mashreduce - server-side mashups for mobile devices. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 6646 LNCS* (2011).

[44] Salz, R. Really simple web service descriptions. `http://www.xml.com/pub/a/ws/2003/10/14/salz.html`, Cited: 15.5.2011, 2003.

[45] Sayre, R. Atom: the standard in syndication. *Internet Computing, IEEE 9*, 4 (2005).

[46] Trifa, V., and Guinard, D. Design of a web-based distributed location-aware infrastructure for mobile devices. *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops, PERCOM Workshops 2010* (2010).

[47] Vinoski, S. Putting the "web" into web services: Web services interaction models, part 2. *IEEE Internet Computing 6*, 4 (2002).

[48] Vinoski, S. Rest eye for the soa guy. *IEEE Internet Computing 11*, 1 (2007).

[49] Walmsley, P., and Fallside, D. C. XML schema part 0: Primer second edition. W3C recommendation, W3C, Oct. 2004. `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028`.

[50] Walsh, N. Witw: Nsdl. `http://norman.walsh.name/2005/03/12/nsdl`, Cited: 15.10.2011, 2005.

[51] Wang, Q., and Deters, R. Soa's last mile connecting smartphones to the service cloud. *CLOUD 2009 - 2009 IEEE International Conference on Cloud Computing* (2009).

[52] Winter, D. Xml-rpc specification. `http://xmlrpc.scripting.com/spec`, Cited: 20.4.2012, 1999.

[53] Work, D., and Bayen, A. Impacts of the mobile internet on transportation cyberphysical systems: Traffic monitoring using smartphones. *National Workshop for Research on High-Confidence Transportation Cyber-Physical Systems: Automotive, Aviation and Rail* (2008).

[54] Yang, Y. C., Cheng, C. M., Lin, P. Y., and Tsao, S. L. A real-time road traffic information system based on a peer-to-peer approach. In *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on* (2008).

[55] Ye, W., Hu, W., Zhao, W., Gao, X., Zhang, S., and Wang, L. Towards lightweight application integration based on mashup. *SERVICES 2009 - 5th 2009 World Congress on Services* (2009).

[56] Yu, J., Benatallah, B., Casati, F., and Daniel, F. Understanding mashup development. *IEEE Internet Computing 12*, 5 (2008).

Appendix A

Client source code

```
<!--
Copyright 2012 Gofore Oy

Web based mobile Digitraffic client.

Author: Hannu Lyytikäinen
-->
<!DOCTYPE html>
<html>
<head>
<title></title>

<meta name="viewport" content="width=480px; height=700px;
  user-scalable=no" />
<meta http-equiv="Content-type" content="text/html; charset=utf-8">

<link rel="stylesheet" href="css/style.css" />
<link rel="stylesheet" href="css/jquery.mobile-1.0.min.css" />
<link rel="stylesheet" href="css/jquery.toastmessage.css" />
<!-- Google Maps API -->
<script src="http://maps.google.com/maps/api/js?sensor=false"
  type="text/javascript"></script>
<!-- jQuery -->
<script src="js/lib/jquery-1.7.1.min.js" type="text/javascript"></script>
<!-- The actual client app -->
<script src="js/traffic.js" type="text/javascript"></script>
<!-- jQuery Mobile -->
<script src="js/lib/jquery.mobile-1.0.min.js"
  type="text/javascript"></script>
<!-- Phonegap -->
<script type="text/javascript" charset="utf-8"
  src="js/lib/phonegap.js"></script>
<!-- jquery-ui-map plugin -->
<script src="js/lib/jquery.ui.map.full.min.js"
  type="text/javascript"></script>
<!-- toast message plugin for notifications -->
<script src="js/lib/jquery.toastmessage.js" type="text/javascript"></script>

</head>
<body>

    <div data-role="page" id="page-map">
```

```
43
    <div data-role="header">
            <a href="" id="refresh" data-role="button" data-icon="refresh">
                Refresh</a>
      <h1>Traffic</h1>
    </div>
48
        <div data-role="content" id="map-content">
                <div id="map_canvas"></div>
        </div><!-- /content -->

53  </div>

</body>

<script>
58  $(traffic.init());
</script>
</html>
```

Example A.1: Client application HTML source code

```
/*
 * Copyright 2012 Gofore Oy
 *
 * Web based mobile Digitraffic client.
5 *
 * Author: Hannu Lyytikäinen
 */

// traffic object that provides map and traffic fluency related
    functionality
10 var traffic = {};

traffic.map = (function() {

     var mapOptions = {
15          'panControl' : false,
            'mapTypeControl' : false,
            'streetViewControl' : false,
            'zoom' : 12,
            'mapTypeId' : google.maps.MapTypeId.ROADMAP,
20          'zoomControlOptions' : {'position' : google.maps.ControlPosition
                .LEFT_BOTTOM,
                'style' : google.maps.ZoomControlStyle.DEFAULT}

        };

25  function locationSuccess(position) {
        var latLng = new google.maps.LatLng(position.coords.latitude,
            position.coords.longitude);

        mapOptions.center = latLng;
30
        // initialize google map
        $('#map_canvas').gmap(mapOptions);
    }

35  function locationError(error) {
        // can't get location, show error
      $().toastmessage('showErrorToast', "Error occured while resoloving
```

```
        location");
    }

    // traffic fluency ajax success call
    function trafficSuccess(data) {
        $('#map_canvas').gmap('clear', 'overlays');

        // iterate the links
        for (x in data) {
            var points = data[x].polyline.points;
            var gLatLngs = new Array(points.length);

            // iterate the points in a link, create
            // new google LatLng onbjects and draw the polyline
            for (y in points) {
                gLatLngs[y] = new google.maps.LatLng(points[y][1],
                  points[y][0])
            }
            var fluencyCode;

            if (data[x].fluencyClass === undefined) {
                // fluency class not found
                fluencyCode = -1;
            }
            else {
                fluencyCode = data[x].fluencyClass.code;
            }

            var colorCode = "";

            // resolve link color from the fluency class
            switch (fluencyCode) {
                case 1:
                    colorCode = "#FF0000";
                    break;
                case 2:
                    colorCode = "#FF8000";
                    break;
                case 3:
                    colorCode = "#FFFF00";
                    break;
                case 4:
                    colorCode = "#0040FF";
                    break;
                case 5:
                    colorCode = "#00FF00";
                    break;
                case -1:
                    colorCode = "#6E6E6E";
                    break;
            }

            $('#map_canvas').gmap('addShape', 'Polyline',
               {
                'editable': false,
                'path': gLatLngs,
                'strokeColor': colorCode,
                'strokeOpacity': 1.0,
                'strokeWeight': 3
                });
        }
    }
```

```
100     function jqxhrError (data, status, xhr) {
          $().toastmessage('showErrorToast', "Error occured while retrieving
              traffic data");
        }

        function jqxhrComplete (data, status, xhr) {
105        $.mobile.hidePageLoadingMsg();
        }

        return {
            init: function () {
110
                $.mobile.loadingMessage = "Loading traffic data";

                // configure toast messages
                $().toastmessage({
115               sticky   : false,
                  position : 'middle-center'
              });

                if (navigator.geolocation) {
120                 navigator.geolocation.getCurrentPosition(locationSuccess,
                        locationError);
                }
                else {
                    // can't get location, show error
                  $().toastmessage('showErrorToast', "Error occured while
                      resoloving location");
125             }
            },

            refresh: function () {
                // retreive traffic data
130             // and draw polylines

                $.mobile.showPageLoadingMsg();

                var bounds = $('#map_canvas').gmap('get', 'map').getBounds();
135             var params =
                    {
                    latMin : bounds.getSouthWest().lat(),
                    lngMin : bounds.getSouthWest().lng(),
                    latMax : bounds.getNorthEast().lat(),
140                 lngMax : bounds.getNorthEast().lng(),
                    includePolylines : "true"
                    }

               var url = "http://www.digitraffic.com/rest/traffic/fluency?
                    callback=?";
145
               var jqxhr = $.getJSON(url, params, trafficSuccess)
                     .success(function() {})
                     .error(jqxhrError)
                     .complete(jqxhrComplete);
150         }
        };
    }());

    traffic.addEvents = function () {
155    $('#page-map').live('pageinit', traffic.map.init);
       $('#refresh').click(traffic.map.refresh);
```

```
};

traffic.init = function () {
    traffic.addEvents();
};
```

Example A.2: Client application JavaScript source code